LEVERAGING EXISTING SOFTWARE ARTIFACTS TO SUPPORT DESIGN, DEVELOPMENT, AND TESTING OF MOBILE APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

By

Farnaz Behrang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2020

LEVERAGING EXISTING SOFTWARE ARTIFACTS TO SUPPORT DESIGN, DEVELOPMENT, AND TESTING OF MOBILE APPLICATIONS

Approved by:

Dr. Alessandro Orso, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Qirun Zhang
School of Computer Science
Georgia Institute of Technology

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Dr. Sam Malek
School of Information and Computer Sciences
University of California, Irvine

Dr. Denys Poshyvanyk
Department of Computer Science
College of William & Mary

Date Approved: June 16, 2020

To my family.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Professor Alessandro Orso, for his guidance and support during my PhD studies. I would like to specifically thank him for giving me the flexibility to choose the projects I wanted to work on, and providing constructive feedback in both my research and presenting my work. I truly enjoyed working with him and have learned a lot from him throughout the years.

I would also like to thank my committee members for their insightful comments and suggestions for improving this dissertation.

Next, I would like to thank the current and past members of the Arktos lab, who helped me navigate the journey of my PhD studies. They have made this journey significantly more fun and tolerable.

And my special thanks goes to my family for their unconditional and endless love, care, and support in every way.

TABLE OF CONTENTS

Acknowledgments iv

List of Tables ix

List of Figures x

Chapter 1: Introduction 1

1.1 Approaches 2

1.1.1 Supporting App Design and Development through GUI Search . . . 2

1.1.2 Automated Test Migration For Mobile Apps 3

1.1.3 Leveraging Execution Traces to Generate Test Inputs 4

1.2 Contributions 5

1.3 Organization 5

Chapter 2: Background and Terminology 7

2.1 Android Platform 7

2.2 Android Components 7

2.3 Testing Mobile Apps 8

Chapter 3: Overview 9

Chapter 4: Supporting App Design and Development through GUI Search . . . 10

4.1 Motivating Example 10
4.2 Technique 12
4.2.1 Analysis Phase 13
4.2.2 Similarity Computation Phase 15
4.3 Evaluation 20
4.3.1 Implementation 20
4.3.2 Analytical Study (RQ1) 21
4.3.3 User Study 22
4.3.4 Discussion and Limitations 30

Chapter 5: Automated Test Migration for Mobile Apps 33

5.1 Motivating Example 33
5.2 Technique 35
5.2.1 Instrumenter 36
5.2.2 Test Runner and Recorder 36
5.2.3 Event Migrator 36
5.2.4 Applying Algorithm 1 to the Example 41
5.2.5 Assertion Migrator 42
5.2.6 Applying Algorithm 2 to the Example 46
5.2.7 Test Encoder 47
5.3 Empirical Evaluation 47
5.3.1 Implementation 47
5.3.2 Evaluation Setup 48

5.3.3 Results 50
5.3.4 RQ1: Accuracy in Migrating Events 52
5.3.5 RQ2: Comparison with GTM 52
5.3.6 RQ3: Accuracy in Migrating Oracles 53
5.4 Threats To Validity 54

Chapter 6: Leveraging Execution Traces to Generate Test Inputs 55
6.1 Technique 55
6.1.1 Instrumenter 56
6.1.2 Recorder 56
6.1.3 Splitter 57
6.1.4 Clusterer 57
6.1.5 FSM generator 57
6.1.6 Test generator 59
6.2 Evaluation 61
6.2.1 Evaluation Setup 61
6.2.2 Results 62
6.2.3 RQ1: Code Coverage 63
6.2.4 RQ2: Crashes 63
6.2.5 RQ3: Usage scenarios 64
6.2.6 Discussion 67

Chapter 7: Related Work 68
7.1 GUI builders 68

7.2 Generating GUI Code from Sketches 68

7.3 Code Search 69

7.4 GUI Test Repair 70

7.5 GUI Test Generation 71

Chapter 8: Conclusion 72

8.1 Future Work 73

8.1.1 Supporting App Design and Development 73

8.1.2 Reusing existing software artifacts within and across programs with similar functionality 74

References 75

LIST OF TABLES

4.1 Apps statistics. 22

4.2 Number of screens in the user sketches and in GUIFETCH recommendations. 23

4.3 Correlation analysis results. 25

5.1 Assertions statistics. 43

5.2 Description of the benchmark apps and tests. 49

5.3 Results of migrating test cases using APPTESTMIGRATOR (ATM) (both events and oracles) and GTM (events only). 51

5.4 Benchmarks assertions statistics. 53

6.1 Benchmark apps. 62

6.2 Results. 63

6.3 Distribution of the detected crashes. 64

6.4 Usage scenarios for *Email Client* category. 65

6.5 Usage scenarios for *Music Player* category. 66

LIST OF FIGURES

3.1 Overview my research. 9

4.1 Sketch of an expense tracking app with two screens and three transitions (Screen A $\xrightarrow{AddNewExpense}$Screen B, Screen B $\xrightarrow[Cancel]{Save}$Screen A). 11

4.2 App returned by GUIFETCH as the top match for the sketch in Figure 4.1 (app transitions: Screen A $\xrightarrow{Addanexpense}$ Screen B, Screen B $\xrightarrow{Save}$ Screen A). 11

4.3 Overview of GUIFETCH. 12

4.4 Overview of the GUI hierarchy generator. 16

4.5 Overview of the transition similarity score calculator. 19

4.6 Number of (relevant) recommendations per screen. 26

4.7 Number of sketch changes per participant. 29

5.1 Sequence of GUI events of a test case that sorts items in a list in the source app. 34

5.2 Sequence of GUI events migrated by APPTESTMIGRATOR from the source app (Fig. 5.1) to the target app. 34

5.3 Overview of APPTESTMIGRATOR. 35

6.1 Overview. 56

SUMMARY

There is an ever growing amount of code available and easily accessible online in public repositories, such as GitHub and Bitbucket. It is therefore not surprising that there has been an increasing interest in analyzing the rich data available in such repositories. Despite the large number of proposed techniques that leverage existing source code, however, these techniques mostly focus on supporting coding and maintenance activities. Other important software engineering tasks, such as software design and testing, have been to a large extent neglected by previous work. To address this limitation of previous work, in this dissertation I defined automated techniques that leverage existing software artifacts to support the design, development, and testing of mobile apps.

Specifically, I defined three techniques: GUIFETCH, APPTESTMIGRATOR, and GUITESTGEN. GUIFETCH is a code-search technique that takes advantage of the growing number of open-source apps in public repositories to support app design and development. Given a sketch of an app's screens and transitions between them, GUIFETCH searches for apps in public repositories that are as similar as possible to the provided sketch, ranks them by similarity to the sketch, and then reports them to the user. GUIFETCH can provide developers with a starting point for building their GUI-based apps, support early prototyping, and help designers assess whether any existing apps are similar to the one they want to develop. APPTESTMIGRATOR is a test migration technique that takes advantage of existing test cases to reduce the cost of testing mobile apps. More precisely, APPTESTMIGRATOR considers similarities between apps and migrates test cases across similar apps. Typical examples of this situation are apps that are developed independently by students as part of a class project or an assignment or apps that belong to the same category, such as banking apps, which share much of their functionality and may provide GUIs that are inherently similar. Finally, GUITESTGEN is a technique that first collects the execution traces generated by one or more apps in a given category when the apps are used by their end-users,

and then leverages the collected traces to generate GUI tests for other apps in that same category (i.e., apps that share part of their functionality).

To evaluate the effectiveness of the techniques I developed, I implemented them as prototype tools and evaluated them through user studies and empirical investigations on real-world apps. My results provide evidence that the techniques are effective in supporting design, development, and testing of mobile apps.

CHAPTER 1
INTRODUCTION

We are living in the era of big data, in which generating and sharing data has become much easier, and massive amounts of information are created in a fraction of a second. In the context of software engineering, in particular, the number of open-source software repositories (e.g., GitHub, Bitbucket, SourceForge) where software developers share their software artifacts is ever-growing, and hundreds of millions of lines of code are freely available and easily accessible. This has resulted in an increasing interest in analyzing the rich data available in such repositories. In the past decade, researchers have been mining online repositories to take advantage of existing source code to support different development activities, such as bug prediction [36], refactoring [66], and API updates [53].

Despite the large number of proposed techniques that leverage existing source code, however, these techniques mostly focus on supporting coding and maintenance activities. There are still many other opportunities that researchers can leverage in terms of extracting useful information produced by developers during the development process and using that information to support a number of additional tasks. Besides source code, many other software artifacts are created, maintained, and evolved as part of the software development process, such as specifications, test cases, and documentation. Researchers can take advantage of these existing software artifacts to support many other development tasks that have been considered only to a limited extent, if at all, before.

In this dissertation work, I specifically focus on leveraging existing software artifacts such as source code, test cases, and execution traces, to support two important software engineering tasks, software design and testing, that have been mostly neglected by previous work. In my work, I focus on mobile applications (or simply apps) due to their importance in our everyday life and their potential for impact on a large number of people.

1.1 Approaches

To support designers and developers of mobile apps by leveraging existing software artifacts, I (1) defined three automated techniques, GUIFETCH, APPTESTMIGRATOR, and GUITESTGEN, (2) implemented these techniques to support Android apps, and (3) evaluated the techniques on real-world apps. In the rest of this section, I present an overview of my techniques.

1.1.1 Supporting App Design and Development through GUI Search

GUIFETCH [8] is a code-search technique that takes advantage of the growing number of open-source apps in public repositories. To design and develop an app, many industrial companies typically go from the sketch of an app to the actual app. The process starts with sketching the graphical user interfaces (GUIs) for the different screens of the app and then creating actual GUIs. Developers then connect the GUIs to the code that implements the app's functionality. To help in part of this process that goes from sketches to GUIs, which involves identifying which layouts to use, which widgets to add, and how to configure and connect the different pieces of the GUIs, GUIFetch takes advantage of open-source apps in public repositories to provide users with GUIs and transitions that are similar to those in their provided sketch.

Given a sketch of an app's screens and transitions between them, GUIFetch searches for apps in public repositories that are as similar as possible to the provided sketch using a combination of static and dynamic analyses, computes a similarity metric between the models and the user provided sketches, ranks the identified apps by similarity to the sketch, and then reports them to the user. GUIFetch can provide developers with a starting point for building their GUI-based apps, support early prototyping, and help designers assess whether any existing apps are similar to the one they want to develop. I empirically evaluated GUIFETCH through user studies and the results show that 81% of GUIFETCH's

recommendations were relevant to the participants' sketches, GUIFETCH's ranking correlated with the ranking provided by the participants in many cases, and participants were able to use GUIFetch's recommendations to make changes to their sketches by adding, updating, or deleting GUI elements.

1.1.2 Automated Test Migration For Mobile Apps

APPTESTMIGRATOR [6, 5] is a technique that migrates test cases between apps that share part of their functionality. It is important to thoroughly test apps to gain confidence that they behave as intended when used in the field. Manually developing test cases for an app tends to be extremely expensive as it involves human effort to define test cases and check test results. I believe the cost of testing apps can be considerably reduced by considering similarities between apps. Although GUIs for different apps can differ dramatically, there are many cases in which apps share similarities that result in conceptually similar GUIs. In these situations instead of creating brand new tests for each app, it should be possible to take advantage of existing test cases and migrate them from one app to other similar apps.

Test migration has several potential applications. It can be used in the educational context to help instructors grade GUI based assignments in very large courses. Most non-trivial assignments that involve the creation of GUI based apps are based on some form of specification, and the students are free to define the GUI of the apps. Grading this type of assignments necessarily requires manual effort, as each app must be checked manually and individually to make sure that it suitably implements the provided specification. Test migration allows the instructor to develop tests for one of the apps and simply migrate them to the other apps. Test migration can also be used to migrate test cases between apps that belong to the same category, such as banking applications, which share much of their functionality and may provide GUIs that are inherently similar. Ultimately, test migration could support the idea of a "test store" that operates in parallel with a traditional app store; when developers submit an app, the test store could analyze the app, look for similar ones,

migrate tests from these apps, and return all the tests that were successfully migrated.

APPTESTMIGRATOR takes as input a source app, a test case for the source app (source test), and a target app, and produces as output the source test migrated to the target app (target test). To do so, it (1) records both the sequence of (GUI) events generated and the assertions checked by the source test, (2) migrates events and assertions to the target app using a similarity metric based on a combination of techniques, and (3) generates a target test case based on the migrated events and assertions. I empirically evaluated the technique and the results show that the technique was able to fully migrate 68% and 48% and partially migrate 21% and 34% of the tests considered in the context of education and general apps, respectively.

1.1.3 Leveraging Execution Traces to Generate Test Inputs

GUITESTGEN is a technique that (1) collects the execution traces generated by one or more apps in a given category when the apps are used by their end-users, and (2) leverages the collected traces to generate GUI tests for other apps in that same category. Unlike APPTESTMIGRATOR, GUITESTGEN does not try to migrate individual tests one by one, but rather it takes advantage of the combination of multiple execution traces that correspond to the same functionality to synthesize tests for that functionality.

The technique takes as input a set of *training apps* and a *target app*, such that all the apps considered belong to the same category. Given this input, the technique produces as output a set of tests for the target app. To do so, it (1) instruments the training apps, (2) records a set of execution traces for these apps, (2) splits and clusters the collected traces, (3) generates a finite state machine (FSM) for each cluster, and (4) uses the generated FSMs to guide the exploration of the target app and generate test inputs for the app. The results of my empirical evaluation of GUITESTGEN show that the technique can, on average and for the benchmarks considered, achieve a slightly higher code and scenario coverage and trigger a considerably higher number of crashes than a random test-input generator. Most

interestingly, in the evaluation, GUITESTGEN and the random test input generator covered a considerably different set of statements and scenarios and triggered a different set of crashes. This provides initial but clear evidence that my technique, by analyzing execution traces of real executions, can generate tests that are different in nature from purely random tests.

1.2 Contributions

This dissertation provides the following novel contributions:

- GUIFETCH: a code-search technique that takes advantage of the growing number of open-source apps in public repositories to provide automated support during the design and development phases of the mobile app development process.
- APPTESTMIGRATOR: a test migration technique that takes advantage of existing test cases to provide automated support during the testing phase of the mobile application development process.
- GUITESTGEN: a technique that leverages the execution traces of a group of apps of a given category to automatically generate tests for other apps in the same category.
- Prototype tool implementations of the techniques.
- Empirical evaluation of the techniques.

1.3 Organization

The rest of this dissertation is organized as follows. Chapter 2 presents background information and terminology related to mobile apps. Chapter 3 provides an overview of how the techniques presented in this dissertation fit in the app development process. Chapters 4, 5, and 6 provide details on GUIFETCH, APPTESTMIGRATOR, and GUITESTGEN, re-

spectively. Chapter 7 describes related work and Chapter 8 provides a conclusion to the research and discusses potential future work.

CHAPTER 2
BACKGROUND AND TERMINOLOGY

This chapter presents background information and terminology related to Android platform, Android components, and testing mobile apps. The background information and terminology introduced in this chapter is used throughout the remainder of this dissertation.

2.1 Android Platform

Android apps are mainly written using Kotlin and Java languages, but there are some platform libraries that allow developers to use C and C++ code with Android to achieve low latency or run computationally intensive apps, such as games or physics simulations. The Android SDK tools [75] compile source code along with any data and resource files into an Android Package Kit (*APK* for short). An *APK* file contains all the contents of an Android app and is used for the distribution and installation of the app.

At runtime, Java classes are converted into DEX bytecode, which is then translated to native machine code via ART or Dalvik, two alternative runtimes. ART was introduced in Android 4.4 (KitKat) and has completely replace Dalvik in Android 5.0 (Lollipop). Android 7.0 added to ART a JIT (just in time) compiler with code profiling to improve the performance of Android apps as they run.

2.2 Android Components

Components are the essential building blocks of an Android app through which the system or a user can enter the app. There are four different types of app components: *Activities*, *Services*, *Broadcast receivers*, and *Content providers*.

Activities represent an app's screens with the user interface. With the help of activities, developers can place all the UI components or elements on an app's screens. Each activity provides a number of callback methods that allow the activity to control the user interactions (e.g., clicks) with the screens.

Services are components that allow apps to perform long-running background tasks or to perform work for remote processes without a user interface.

Broadcast receivers are components that allow registering for system or app events. All registered receivers for an event are notified by the Android runtime once the event happens. For instance, apps can register for the system event, which is fired once the device starts charging.

Content providers are components that act as a central repository to store the app's data and allow other apps to access the data if necessary. For instance, the Android system manages the user's contact information through a content provider that can be queried by any app with the proper permissions.

2.3 Testing Mobile Apps

Mobile apps are largely tested through their GUIs; GUI-based testing aims to bring the app into a particular state through a sequence of GUI events, such as clicking on a button or submitting text to a form, and uses oracles to check the outcome of the test (e.g., the existence or the specific value of a property for a given GUI element). Since oracles in this context are assertion-based, I use terms *oracle* and *assertion* interchangeably. A *GUI state* s is a set of triples (e, p, v), where e is an element on the screen, p is a property of w, and v is the value of p. A *GUI event* e is a triple (a, t, i), where a is the action that corresponds to the event (e.g., click), t is the target of the action (e.g., button), and i is the (optional) input value (e.g., data for a text input box). An *assertion* as, is a function $F : (e, c) \rightarrow \{True, False\}$, where *e* is a GUI element, and *c* is a condition to be checked for that GUI element. The function returns *True* if *e* satisfies condition *c*, and *False* otherwise.

CHAPTER 3

OVERVIEW

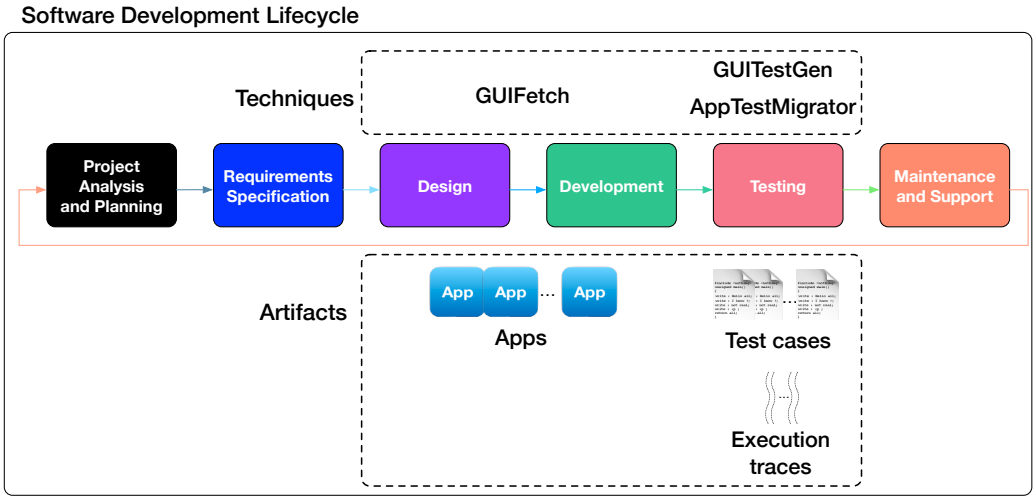

Figure 3.1: Overview of how my techniques fits in the app development process.

Figure 4.3 illustrates a high-level overview of my research. All of the techniques leverage existing software artifacts such as apps, test cases, and execution traces to support design, development, and testing of mobile apps. GUIFETCH is a code-search technique that takes advantage of the growing number of open source apps in public repositories to provide automated support during design and development phases of the app development process. APPTESTMIGRATOR is a test migration technique that takes advantage of existing test cases and GUITESTGEN is a technique that leverages the execution traces of human users to provide automated support during testing phase of the app development process. Chapters 4, 5, and 6 describe the techniques in detail.

CHAPTER 4

SUPPORTING APP DESIGN AND DEVELOPMENT THROUGH GUI SEARCH

This chapter presents GUIFETCH, a code-search technique that takes advantage of the growing number of open source apps in public repositories to support designers and developers of mobile apps. The work described here was originally published in [8]. The rest of this chapter is structured as follows. Section 4.1 presents a motivating example, Section 4.2 details the technique, and Section 4.3 discusses the evaluation of GUIFETCH.

4.1 Motivating Example

As an example, assume a developer or a GUI designer uses a sketching tool to draw the sketch shown in figure 4.1, which represents an expense tracking app. The sketch consists of two screens with three possible transitions (shown with arrows in the sketch). The first screen shows the total expenses for the current month and has two buttons: *View Expense* and *Add New Expense*. Clicking the *Add New Expense* button opens the second screen, which allows the user to enter the expense details, such as date, description, cost, and type. Clicking on either *Cancel* or *Save* brings the user back to the first screen.

Given a set of keywords and the sketch, GUIFETCH would search in one or more open source code repositories and return to the user a set of relevant apps. To illustrate, Figure 4.2 shows the screens of an app that was returned by GUIFETCH as the top match for this sketch in the evaluation. As the figure shows, the app also contains two screens. Moreover, similar to the sketch, clicking on button *Add an expense* takes the users to the second screen and allows them to enter the details of an expense. Clicking on the *Save* button, conversely, causes the app to go back to the first screen.

GUIFETCH would return the screenshots of the app (shown in figure 4.2), along with the app source code. Providing the screenshots, in addition to the source code, makes it eas-

Figure 4.1: Sketch of an expense tracking app with two screens and three transitions (Screen A $\xrightarrow{AddNewExpense}$ Screen B, Screen B $\xrightarrow[Cancel]{Save}$ Screen A).

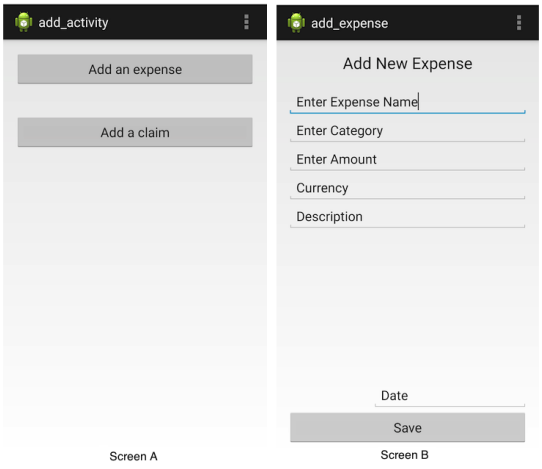

Figure 4.2: App returned by GUIFETCH as the top match for the sketch in Figure 4.1 (app transitions: Screen A $\xrightarrow{Addanexpense}$ Screen B, Screen B $\xrightarrow{Save}$ Screen A).

ier for the users to perform a first screening of the results, without any need for compiling or running the code.

Figures 4.1 and 4.2 clearly show that, although the sketch and the app are not identical, they are conceptually fairly similar. In particular, there is a mapping between the elements of the sketch and the widgets of the app. Also, two out of three transitions in the user provided sketch exist in the app.

It is worth noting that, in addition to the similar elements that exist in both the sketch and the app, there are two more elements in the app—related to expense name and currency—that do not exist in the sketch. If the designer or the developer found these elements relevant for what they had in mind, they could add them to their design by just looking at the screenshots. They would also have the opportunity to take a look at the source code to see if they are interested in using it as an initial prototype or as a starting point for building the app.

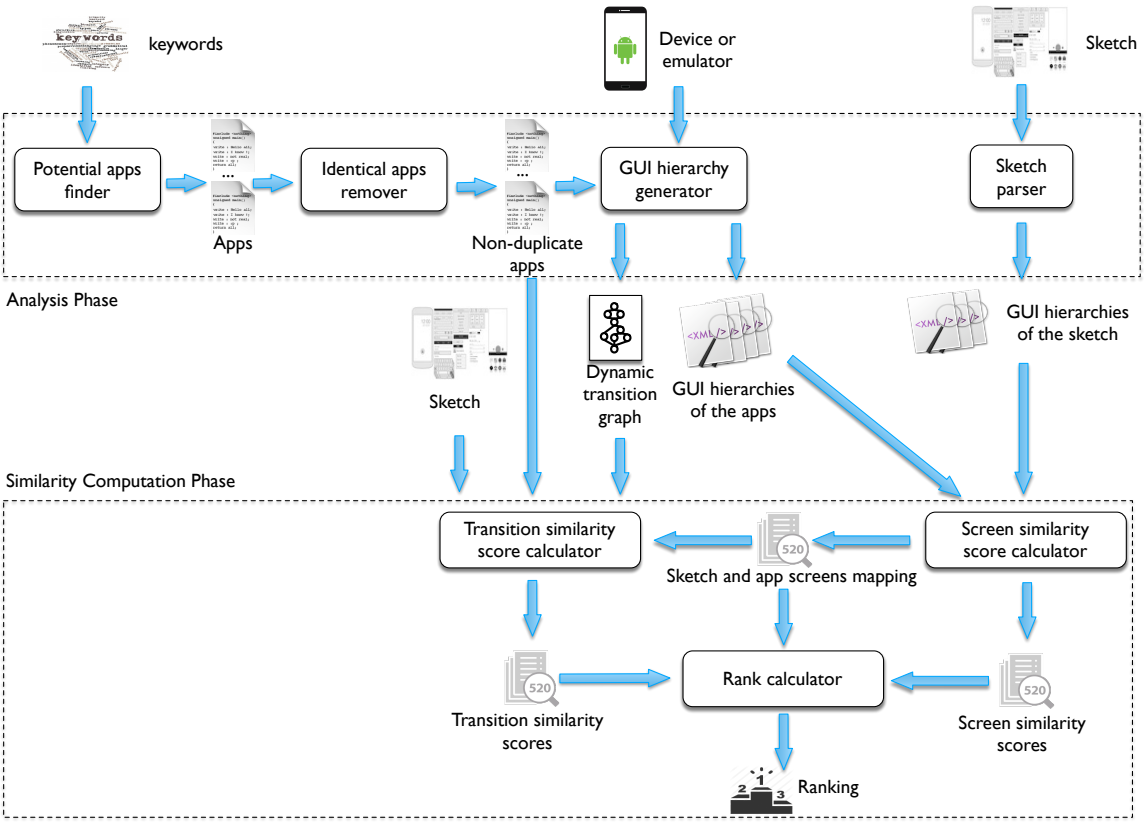

Figure 4.3: Overview of GUIFETCH.

4.2 Technique

Figure 4.3 shows an overview of the technique, GUIFETCH. As the figure shows, the technique is divided into two phases: *Analysis* and *Similarity Computation*.

Given a set of keywords and a sketch of an app, the *Analysis Phase* is responsible for finding the potential apps from open source code repositories, removing duplicates, and analyzing the source code and the sketch. The outputs of this phase are GUI hierarchies for every possible screen of the apps and the sketch, together with transition graphs for the apps. The *Similarity Computation Phase* uses these generated GUI hierarchies and transition graph to (1) compute a similarity score between the sketch and the apps and (2) provide a ranking based on the computed similarity scores. In the rest of this section, I explain each phase in more detail.

4.2.1 Analysis Phase

Potential Apps Finder

The first stage of the analysis phase involves finding relevant apps in an open-source repository given a set of keywords related to the app of interest.

The search process starts with the keywords and performs two separate searches. The first search looks for Java source files containing the keywords as well as the term "Android". Source files are considered here because they are more likely to contain comments that describe the application and that might be a good match for the user-provided keywords. The second search looks for Android manifest files that contain the keywords. More precisely, this search focuses on XML files that contain the original keywords as well as the terms "Android", "manifest", "application", and "activity". For each of these searches, GUIFETCH looks at the first 100 matching files returned by GitHub. (I currently use GitHub as the search engine since the underlying repository includes not only the source files but also all the related Android resource files which will be needed to run the program.) GUIFETCH treats each returned file as an indicator of what project should be considered. It then tries to compile the corresponding projects and considers as potential apps those that compile successfully.

Identical Apps Remover

Retrieved code from open source repositories is likely to include identical or highly similar apps. GUIFETCH removes duplicates to reduce the running time of the technique and make the final ranking more useful to users. To do so, GUIFETCH uses an existing plagiarism detection technique [64]. Although this technique does not consider files other than source code (*e.g.,* XML files), it is highly unlikely for two apps to have extremely similar GUIs and different source code. GUIFETCH removes all the apps with a similarity value higher than 70%, a threshold computed based on preliminary experiments.

Sketch Parser

To make the user sketch comparable to real apps, GUIFETCH needs to generate GUI hierarchies for it, along with transitions between the GUIs. The first step in generating comparable GUI hierarchies is to find a mapping between the types of Android widgets and the type of elements in the sketch. For screenshots or conceptual drawings, for instance, this is achievable using OCR and computer vision techniques [58], whereas various heuristics can be used for SVG-based sketches [71]. For some prototyping tools, in particular, it might be possible to directly parse their output, as I do in the current implementation (see Section 5.3.1.

After finding the mapping between widget types, GUIFETCH extracts all the elements of the sketch along with their attributes. These attributes include type, height, width, dimensions, and any text associated with the widgets. (For dimensions and coordinates, values relative to the sketch size are used.) Then, GUIFETCH extracts from the sketch the transitions between screens provided by the user and builds a graph where nodes are screens and edges are transitions labeled with the widgets that cause the transition.

GUI Hierarchy Generator

After removing identical apps and parsing the sketch, GUIFETCH generates the GUI hierarchies for every possible screen of the app. In Android, GUIs can be built both statically and dynamically. Specifically, GUIs (or parts thereof) can be built statically through XML layout files or dynamically through calls to library functions. Therefore, to identify all screens and their corresponding widgets, GUIFETCH needs to perform both static and dynamic analyses; (1) a purely static technique would miss those parts of the GUI that are built dynamically, whereas (2) a purely dynamic tehcnique may not be able to reach, and thus model, all screens and widgets therein.

The overview of the *GUI Hierarchy Generator* is shown in Figure 4.4. GUIFETCH first gets the source code of the app and finds a mapping between screen layout files and source

files. In this way, it identifies all the possible screens of the app and detects which layout and source files correspond to which screen. Note that, although this analysis identifies all screens, the widgets in the screens can be modified at runtime. For this reason, the GUI hierarchy that is obtained dynamically, when the dynamic analysis is successful, is generally more precise than the one computed statically.

After identifying screens and mappings to layout and source files, GUIFETCH performs a dynamic analysis based on crawling—it launches the app and generates events, such as clicks, long clicks, and so on, through the user interface trying to reach all the screens in the app. Every time a new screen is observed, GUIFETCH records it. Since transitions between screens also matter, GUIFETCH keeps track of such transitions during the analysis. Finally, GUIFETCH models the execution of the app as a transition graph where nodes represent screens and edges represent transitions between screens.

The outputs of the dynamic crawler are the GUI hierarchies of each screen identified and a graph of transitions between them. GUIFETCH then compares the set of screens identified dynamically with the set of all screens determined initially. For every screen that was not reached dynamically, it uses a static renderer that generates a screen based on the corresponding layout file and gets the GUI hierarchy for that screen. The final output of this step is the combination of the GUI hierarchies collected statically and dynamically.

4.2.2 Similarity Computation Phase

In this phase, GUIFETCH takes as input the source code of non-duplicate apps, the GUI hierarchies of the screens of these apps and of the sketch, and the dynamic transition graph it computed in the analysis phase. Given this input, GUIFETCH computes an *overall similarity* score between each app considered and the sketch, as follows. First, GUIFETCH computes the similarity between each screen in the app and each screen in the sketch (*screen similarity*, see Section 4.2.2). Second, GUIFETCH defines a mapping between screens in the app and screens in the sketch based on their corresponding screen similarity score. It

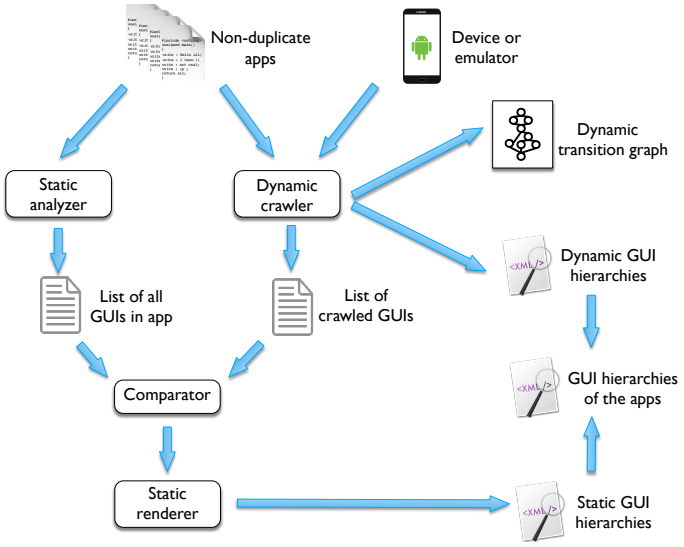

Figure 4.4: Overview of the GUI hierarchy generator.

then uses this mapping to compute a measure of how well the transitions in the app and in the sketch match (*transition similarity*, see Section 4.2.2). Finally, GUIFETCH adds the various screen similarity scores (for all the screens that match) and the transition similarity score to compute the overall similarity score. I now describe in more detail these steps.

Screen Similarity Score Calculator

The inputs to this step are two GUI hierarchies: one for a screen of the sketch and the other for a screen of an app. GUIFETCH compares these GUI hierarchies by widget and by considering four criteria for every widget:[1] type, associated text (if any), size (width and height), and position. We choose these four criteria as they were shown to be effective in earlier work [71]. In the rest of this section, I discuss how each of these criteria affects the similarity score of two widgets. Note that the percentages that we assign to the different criteria are based on the preliminary empirical investigation, in which I trained GUIFETCH on about 100 apps and tested it on 40 different apps to identify effective thresholds.

GUIFETCH performs *type matching* between sketch widgets and app widgets using the

[1]For ease of explanation, hereafter I refer to both the actual widgets in the app and the elements in the sketch as *widgets* and use the terms *app widget* and *sketch widget*, respectively, to refer to them.

mapping between sketch elements and Android widget types it computed while parsing the sketch (see Section 4.2.1). For each sketch widget, GUIFETCH considers all potentially matching app widgets and computes a similarity score as follows. If both widgets contain text, GUIFETCH performs *text matching*; Otherwise, it moves to the size and position matching. To perform text matching, GUIFETCH first applies some heuristics that aim to improve precision. Specifically, GUIFETCH uses a POS (Part-Of-Speech) tagger to assign POS tags to each word in the text, such as noun, verb, adjective, and so on. In this way, GUIFETCH can ignore irrelevant elements, such as pronouns, conjunctions, and prepositions, when comparing text. For instance, a header in a sketch screen with title "AddressBook" would match the header in an app screen with title "My AddressBook", as the pronoun "My" would be ignored. GUIFETCH also converts all the characters in the text to lower case. After this preprocessing, GUIFETCH computes the Levenshtein distance [3] between the two text elements, assigns a score to this match accordingly, and checks whether the score is above a given threshold. If not, GUIFETCH sets the similarity score to zero and stops considering the pair as a possible match. Otherwise, it normalizes the score to 60 (*i.e.,* text matching constitutes 60% of the overall similarity score) and continues. (I chose this value based on preliminary experimentation, as I discussed earlier in this section.)

In the *size and position matching*, GUIFETCH first normalizes the width, height, x coordinate, and y coordinate of app and sketch widget relatively to the screen size. Then, it computes the differences for each of this four normalized values between the two widgets, computes four separate scores proportional to such differences, and normalizes each score to 10 (*i.e.,* each score constitutes 10% of the overall similarity score). Also in this case, if the resulting similarity is below a given threshold, GUIFETCH sets the similarity score to zero and stops considering the pair as a possible match. Otherwise, it either adds the computed similarity to the text matching similarity or further normalize the similarity score to 100, if no text matching was performed.

After all the potentially matching app widgets for a sketch widget have been evaluated, GUIFETCH selects as a match the app widget that has the highest similarity score. Finally, when all the widgets on the input sketch screen have been processed, GUIFETCH computes the overall similarity score between this screen and the input app screen as the ratio consisting of the sum of the similarity values for all the sketch widgets over the maximum possible value for the sum (*i.e.,* 100 times the number of sketch widgets).

Transition Similarity Score Calculator

As I mentioned earlier, besides measuring the similarity of screens, GUIFETCH also computes a similarity score for the transitions between screens. Figure 4.5 shows an overview of how GUIFETCH computes such *transition similarity score*.

The analysis phase of GUIFETCH (see Section 4.2.1) produces a graph of screens and transitions derived from the dynamic analysis. To compute transition similarity scores, GUIFETCH complements this dynamically computed information with information computed through static analysis. Specifically, the technique uses static analysis to compute a static transition graph and then merges these the dynamic and static graphs into a single graph. To merge the graphs, GUIFETCH adds all the nodes and transitions of the dynamic transition graph to the static transition graph if the nodes or transitions do not already exist. To compare nodes and check for their existence, GUIFETCH compares the GUIHierarchy of the corresponding screens. Conversely, GUIFETCH compares transitions by comparing their labels, which correspond to the widgets that caused the transitions. After merging dynamic and static transition graphs, GUIFETCH compares the merged graph with the corresponding transition graph derived from the sketch. The goal of this comparison is to check whether the transitions defined by the user in the sketch exist in the app. Therefore, for every transition between two nodes of the sketch transition graph, GUIFETCH checks if such a transition exists in the app transition graph, using the mapping between screens in the app and screens in the sketch determined in section 4.2.2. If the transition exists,

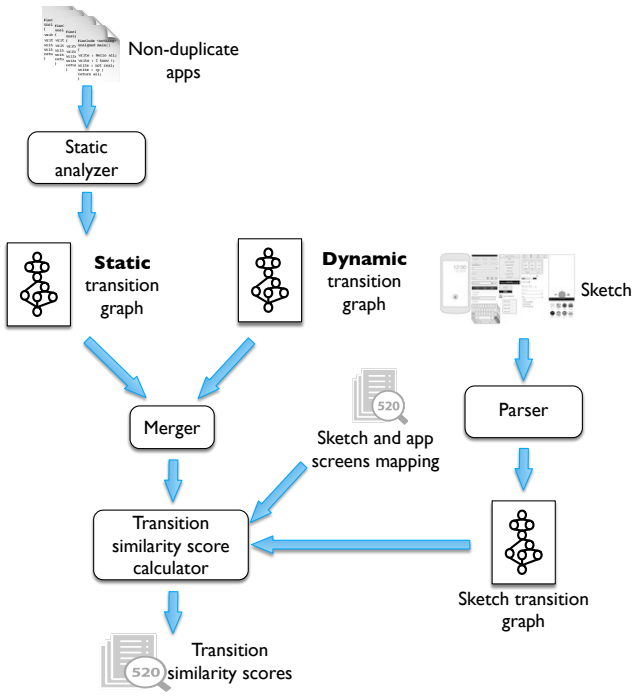

Figure 4.5: Overview of the transition similarity score calculator.

GUIFETCH assigns the (previously computed) similarity score for the two widgets that cause the transition in the two graphs as the similarity score for the transition. Conversely, if the transition does not exist, its similarity score is considered to be zero.

Rank Calculator

After computing screen and transition similarity scores for all the screens and transitions in the sketch, GUIFETCH (1) computes the overall similarity score by adding these scores and (2) ranks the apps based on this value. It is worth noting that the ranking at the app level might not be effective in all cases. Suppose, for instance, an app that has one screen with a very low (or a very high) screen similarity score compared to other screens' similarity scores. In this case, the similarity score for this screen could dramatically change the ranking of the app. To make users aware of this possible effect, besides showing the ranking at the app level, GUIFETCH also provides the ranking of each app's screen per sketch screen. In this way, users can see both rankings, side by side and decide which one works

better for them.

Since the keyword search can return apps that are completely irrelevant or simply too different from the sketch, it is important to identify and discard these apps, which would only introduce noise and overwhelm the user. To do so, GUIFETCH keeps track of the difference between every two consecutive overall similarity scores in the ranking and stops the ranking at the point where the largest gap occurs.

4.3 Evaluation

To assess effectiveness and usefulness of the approach, I developed a tool that implements GUIFETCH and used the tool to investigate four research questions:

1. **RQ1:** Are GUIFETCH recommendations accurate?
2. **RQ2:** How relevant are GUIFETCH recommendations for users?
3. **RQ3:** How well does GUIFETCH ranking match user ranking?
4. **RQ4:** Do GUIFETCH recommendations provide users with insights on how they could improve their initial sketches?

I investigated RQ1 through an analytical study (Section 4.3.2) and RQ2, RQ3, and RQ4 through a user study. In the rest of this section, I describe in detail the implementation, the empirical study, and the user study.

4.3.1 Implementation

The implementation of GUIFETCH supports Android applications. I chose Android because it is one of the major platforms in the mobile application market and because there is a large number of open source apps available for this platform. The approach, however, is general and could be implemented for other platforms.

To implement the apps search, I leveraged the open source framework S^6 [70], which I modified to better handle searches involving Android apps. In particular, I added to S^6 the ability to search for manifest files, in addition to source code, and interpret the manifest

contents in terms of source files referenced therein.

To support app sketches with multiple screens and screen transitions, I implemented in the tool support for *Pencil* [61], an open-source GUI prototyping tool. *Pencil* can be run as a standalone application or within the Firefox browser, as a plugin. It provides 73 different Android specific widgets and allows users to define transitions from any widget to any screen. Although the sketch can be exported in several formats, I chose to parse the *Pencil* document file, which is in XML format, the tool could easily be extended to support other formats or other kinds of sketches.

I implemented the dynamic analysis on top of Espresso [20], a testing framework that provides APIs for writing GUI tests that simulate user interactions with an app. The dynamic crawler is inspired by PUMA [28], which is a programmable GUI-automation framework for dynamic analysis. For the static analysis, I leveraged gator [85], a static analysis tool that creates a model of the GUI-related behavior of an Android app, as well as a model of the app's control flow. I used gator to find the mappings between layouts and source files and to generate static transition graphs for an app. I also leveraged the rendering engine of Android Studio [50] to render an app's layout files and retrieve the corresponding GUI hierarchies.

4.3.2 Analytical Study (RQ1)

To answer RQ1 I studied whether, given a sketch (S) and a set of apps one of which (A) matches the sketch, GUIFETCH would successfully recommend A as the best match for S. Specifically, I selected six different types of apps, namely, address book, expense tracking, note taking, fitness, mobile banking, and online shopping apps. I chose these categories because they represent a diverse range of apps and because are well represented in GitHub (*i.e.,* they are good candidates for evaluating GUIFETCH's ability to find and rank apps). Table 4.1 provides, for each of the app categories that I studied (Column "App category), the total number of apps that GUIFETCH found for that category through keyword search

Table 4.1: Apps statistics.

App category	Apps#	Scr#/Trans# (App 1)	Scr#/Trans# (App 2)
address book	29	1/0	2/2
expense tracking	28	3/6	4/4
note taking	25	2/4	3/6
fitness	22	5/11	6/6
mobile banking	18	4/7	6/8
online shopping	23	5/5	7/11
Total	145	-	-

and after eliminating duplicates (Column "App#"). As keywords, I simply used the name of the categories along with the word "Android".

To create the sketches needed for the study, I randomly selected two apps for each app category considered among the ones I found, ran them manually to identify their screens and transitions, and created their sketches accordingly using the *Pencil* tool. The two columns labeled "Scr#/Trans#" in Table 4.1 report the number of screens and transitions for the two randomly selected apps.

For each sketch S I created for each category C, I provided S to GUIFETCH and ran it against all the apps in C, including the app randomly selected to generate the sketch. For all twelve sketches, GUIFETCH ranked the correct app as the top match for the sketch. To further evaluate GUIFETCH's performance, I also confirmed that, for every screen of the sketch, the corresponding screen in the correct app was always the top match.

4.3.3 User Study

This section describes the user study I conducted to answer the rest of the research questions.

Table 4.2: Number of screens in the user sketches and in GUIFETCH recommendations.

User ID	*Category*	*#Sketch Screens*	*#Recomm. Screens*
1	address book	2	1
2	address book	3	2
3	address book	3	2
4	address book	1	1
5	address book	1	0
6	address book	4	2
7	address book	2	2
8	address book	2	1
Total	-	**18**	**11**
9	mobile banking	7	4
10	mobile banking	4	3
11	mobile banking	5	3
12	mobile banking	3	1
13	mobile banking	4	3
14	mobile banking	3	3
15	mobile banking	5	3
16	mobile banking	5	4
Total	-	**36**	**24**
Total	-	**54**	**35**

Participants

I recruited 16 participants (6 male, 10 female, aged 19 to 30). The participants were recruited by advertising the study in undergraduate- and graduate-level HCI related courses. I required participants to have either Android app design or Android app development background. 60% of the participants had the former, whereas 40% had both. All participants were given a $25 gift card for their participation.

Protocol

At the beginning of the study, each participant was randomly assigned a category of mobile apps that was either "address book" or "mobile banking". I chose these two categories, also randomly, among the categories I considered in the analytical study (see Section 4.3.2). I limited the number of categories so as to have an adequate sample size in each category

and be able to run statistical tests [15]. I then introduced the participants to the *Pencil* sketching tool and gave them some time to become familiar with the tool and possibly ask us questions about it. After making sure that the participants were comfortable with the tool, I asked them to start drawing the sketch of an Android app related to their randomly assigned category. Participants were allowed to use any resources when drawing the sketch (*e.g.,* online search or apps on their phone). I also provided them with a pencil and a sheet of paper, in case they wanted to draw the sketch on paper first or needed to take notes.

Once participants drew the sketch, they were asked to run the GUIFETCH tool with their sketch as input. To reduce the waiting time associated with a real-time GitHub search, in the study I connected GUIFETCH with a local database that contained the apps GUIFETCH found, for the categories considered, in the context of the analytical study (see Section 4.3.2). As shown in Table 4.1, this set consisted of 29 address book and 18 mobile banking apps.

After running GUIFETCH and receiving the recommendations it produced, participants were asked to separate the recommendations they found to be relevant for their sketches from the ones they considered irrelevant. Participants were also asked to manually rank the recommendations they received, so that I could compare their rankings with the rankings generated by GUIFETCH.

To support these tasks, I developed a web application designed to guide the participants through the different steps of the study. The application provided instructions for each task to be performed, let the participants perform the task from within the application, and recorded the results of the task. Specifically, when participants first started using the web application, the application provided them with information on the purpose of the study. An example for each task was also shown to the participants to make sure they understood what they had to do.

To perform the tasks related to the relevance of the recommendations, participants were shown, for each screen of their sketch and in a random order, all the recommendations

Table 4.3: Correlation analysis results.

Screen#	*Kendall's Tau coeff.*	*Kendall's Tau p*	*Spearman coeff.*	*Spearman p*
1	0.6	0.142	0.8	0.104
3	1	-	1	-
5	0.733	0.039	0.829	0.042
6	1	-	1	-
7	1	-	1	-
8	1	-	1	-
9	1	-	1	-
10	0.714	0.024	0.857	0.014
11	0.8	0.05	0.9	0.037
12	1	-	1	-
13	1	-	1	-
14	1	-	1	-
18	0.867	0.015	0.943	0.005
20	1	-	1	-
21	0.8	0.05	0.9	0.037
22	0.733	0.039	0.829	0.042
23	0.6	0.142	0.7	0.188
25	1	-	1	-
26	1	-	1	-
28	0.667	0.174	0.8	0.2
29	1	-	1	-
30	1	-	1	-
32	1	-	1	-
34	0.8	0.05	0.9	0.037

generated by GUIFETCH for that screen. They were then asked to mark, using a checkbox, the recommendations they thought were irrelevant for that particular screen of their sketch.

To perform the tasks related to ranking the recommendations, rather than simply presenting all the recommendations sequentially and asking the participants to rank them, I asked for a set of pairwise comparisons. The rationale for this choice is that the number of recommendations depends on the participant's sketch and can be high. Due to the limitations of human short-term memory, remembering and comparing a potentially large number of recommendations shown in sequential order is difficult and error-prone. (I confirmed this issue in the pilot study, in which the participants were annoyed when they had to rank more than a few recommendations at a time.) With pairwise comparison, conversely, participants compare only two recommendations at a time, and I can still derive an overall

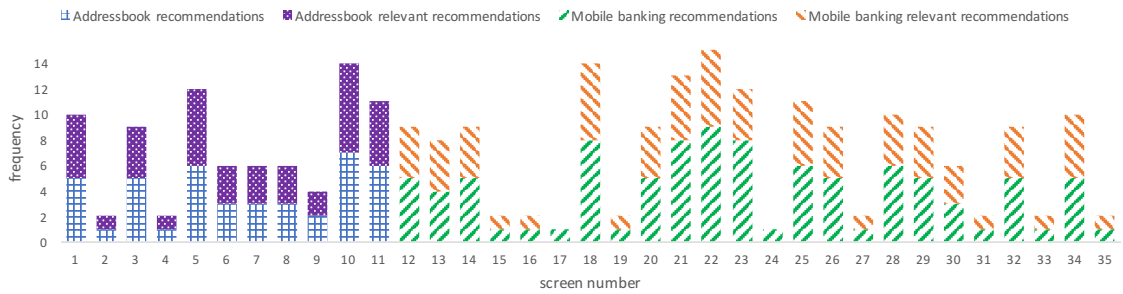

Figure 4.6: Number of (relevant) recommendations per screen.

ranking from the comparisons of the individual pairs.

After performing the above tasks, participants were asked whether they wanted to update their sketch based on GUIFETCH's recommendations. If so, they could visualize the recommendations again while they were working on their sketch. I also explicitly mentioned to the participants to try as much as possible to avoid changes that they wanted to perform for reasons other than seeing the recommendations.

Finally, I asked the participants to fill out a questionnaire about their background and their feedback on GUIFETCH, based on their experience with it. In particular, the questionnaire asked participants about their experience with Android. It then asked them if GUIFETCH's recommendations provided them with any insight or inspiration to update their initial sketch. If so, the questionnaire also asked which specific recommendations they used and why? Next, the questionnaire asked their opinions about the potential applications and extensions of GUIFETCH. Finally, it gave them the option to provide any additional comments about their experience with GUIFETCH.

GUIFetch Recommendations

Table 4.2 shows information about the sketches drew by participants and GUIFETCH's recommendations for those sketches. The first column shows the participant id, followed by the name of the assigned category, the number of screens in the sketch, and the number of screens for which GUIFETCH produced at least one recommendation. The number of

sketch screens ranges from 1 to 7, with mean, median, and variance of 3.375, 3, and 2.65, respectively. The number of screens for which GUIFETCH produced at least one recommendation ranges from 0 to 5, with mean, median, and variance of 2.1875, 2, and 1.3625, respectively. Note that both the mean and the total number of screens for the *address book* category are less than those for the *mobile banking* category, as *address book* apps generally provide less functionality compared to *mobile banking* apps. In both categories, the participants drew 18 and 36 screens, and GUIFETCH was able to find recommendations for 11 (61%) and 24 (67%) of them, respectively. In total, GUIFETCH produced at least one recommendation for 35 out of 54 screens (65%).

Relevance of GUIFetch Recommendations (RQ2)

Figure 4.6 shows the total number of GUIFETCH recommendations and the number of these recommendations that the users considered to be relevant. The results are presented for the 35 screens for which GUIFETCH produced at least one recommendation.

As the figure shows, participants found the recommendations for 19 of the 35 screens (54%) to be 100% relevant. Additionally, the participants found 80% or more of the recommendations to be relevant for 8 screens (23%), and between 60% to 80% of the recommendations to be relevant for 6 screens (17%). For the remaining two screens, GUIFETCH produced a single recommendation, which the participant found to be irrelevant. Overall, of the 138 recommendations provided by GUIFETCH for 35 screens, participants found 112 recommendations (81%) to be relevant to their sketches. It is worth noting that 42 of the 138 recommendations are for the *address book* sketches, with 40 of them (95%) classified as relevant by the users, whereas the remaining 96 are for the *mobile banking* category, with 72 of them (75%) classified as relevant.

Matching of GUIFetch and User Ranking (RQ3)

To assess the recommendations ranking provided by GUIFETCH, I measured how well GUIFETCH's ranking matched the participants' ranking using two well-established measures of non-parametric rank correlations: Kendall's tau [35] and Spearman's rho [77]. Kendall's tau is calculated based on the number of pairwise agreements between two ranking lists, and is computed using the formula $\frac{n_c - n_d}{N}$, where n_c is the number of concordant (ordered in the same way) pairs, n_d is the number of discordant (ordered differently) pairs, and N is the total number of pair combinations. Spearman's rho is calculated based on the difference in the ranking positions of each item in the ranking, and is computed using the formula $1 - \frac{6 \times \sum_1^n d_i^2}{n(n^2-1)}$, where d_i is the difference in paired ranks, and n is the number of items. I ran correlation analysis using both measures given the null hypothesis of no correlation between the rankings by GUIFETCH and by the participants.

Table 4.3 shows the results of the correlation analysis. The first column in the table shows the screen number (same as figure 4.6), followed by Kendall's Tau's correlation coefficient and p-value, and Spearman's correlation coefficient and p-value. Out of the 24 screens for which GUIFETCH produced more than one recommendation, and thus a ranking, user and GUIFETCH rankings were identical for 14 screens (58%). For these rankings, the correlation coefficients for both measures are therefore one. For the remaining 10 screens (42%), Kendall's Tau's coefficients range from 0.6 to 0.867, while Spearman's coefficients range from 0.7 to 0.943. For 4 screens, the p-values for both Kendall's and Spearman's coefficients are less than 0.05, which means that the results are statistically significant, and I can reject the null hypothesis. For 3 screens, Spearman's p-values are less than 0.05 (*i.e.,* statistically significant), while Kendall's p-values equal 0.05, thus weakly rejecting the null hypothesis. For the remaining 3 screens, both Kendall's and Spearman's p-values are greater than 0.05, which means that, in these cases, there is not enough evidence to reject the null hypothesis; I can therefore conclude that there is no correlation between the GUIFETCH and user rankings for these 3 screens.

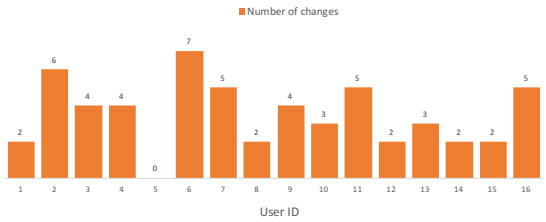

Figure 4.7: Number of sketch changes per participant.

Insights from GUIFetch Recommendations (RQ4)

To evaluate whether GUIFETCH's recommendations provided any insights to participants on ways to update their initial sketches, I measured the number of changes they made to the sketches after seeing these recommendations. I also examined the feedback the participants provided in the questionnaire I asked them to complete.

I define a *change* as the action of adding, deleting, or updating any widget in the sketch. In the case of identical changes performed on multiple screens (*e.g.,* adding a sign-out button to all screens), I counted the change only once. In addition, I manually confirmed that every change performed was based on a recommendation from GUIFETCH. Based on this definition, figure 4.7 shows the number of changes made to the sketches by each participant. As the figure shows, the number of changes ranges from 0 to 7, with mean, median, and variance of 3.5, 3.5, and 3.333, respectively. Out of 56 changes, 47 changes (84%) consisted of adding new widgets to the sketches, 5 changes consisted of updating an existing widget (9%), and 4 changes (7%) consisted of deleting a widget.

All participants but one—the same participant who also provided no recommendations in Table 4.2—answered "Yes" to the question of whether GUIFETCH recommendations provided them with any new insight. Based on the answers in the questionnaire, I identified three common ways in which GUIFETCH's recommendations helped the participants. (1) Reminders of basic, yet necessary GUI elements. Participant 10, for instance, added 3 GUI elements to his initial sketch and explained the change as follows: *"In a specific screen, I sometimes missed some essential components of the screen. So I could modify my screen*

design by reflecting GUIFETCH recommendations." (2) Showing of design alternatives. Participant 3, for example, performed 4 updates to his initial sketch and elaborated: *"The fields in my screen were already similar to images X and Y. However, they were all text based. The image Z was a good visual representation of the app, and I just remembered the importance of images to represent information. Thus, in the refined design, I updated the text fields to icons"*. (3) Inspiring new ideas. Participant 13, for instance, added 3 GUI elements that were not in the recommendations, but were inspired by them, as the participant described: *"It helped with brainstorming and revising. It allowed me to see others' work, which helped to generate new ideas and missing aspects."*.

4.3.4 Discussion and Limitations

As the results of the experiments show, the participants do not find all the recommendations produced by GUIFETCH to be relevant or useful. And the rankings provided by GUIFETCH do not always match the user rankings. However, the results provide clear evidence that GUIFETCH recommendations can be useful to users in a vast majority of cases. Besides, I speculate that, due to the somehow subjective nature of app matching, there is no one-size-fits-all possible ranking, and there may be an inherent upper bound on how accurate a technique can be. Despite this, in future work I plan to further analyze the results to see whether I can identify ways to further improve the technique.

To get a better understanding of possible applications and limitations of GUIFETCH, I added to the questionnaire two follow-up questions on potential applications and extensions of GUIFETCH and also gave the participants the ability to provide general feedback about the tool. A participant mentioned that GUIFETCH might be used to do some quick design for throw-away, low-fidelity prototyping. Another participant thought that GUIFETCH could serve as a useful checklist for checking the inclusion or exclusion of features and getting confidence that a consistent UI experience is provided to the user (akin to the concept of best practices). Yet another participant found GUIFETCH useful in helping users think out-

side the box. Two other participants mentioned a new possible application for GUIFETCH that I had not yet considered: using the tool for learning purposes. Specifically, one of them indicated that GUIFETCH could help novice designers to learn more quickly, as it would allow them to compare their designs with the recommended ones, possibly finding flaws and improving the designs accordingly. Finally, one participant mentioned his experience taking an introductory mobile apps development course, in which the students struggled to find initial code to use as a starting point; he believed that having GUIFETCH would have helped the students.

Regarding possible extension of GUIFETCH, one of the participants mentioned that, since designers in digital space are dependent on applications such as Photoshop, Illustrator, Sketch, and InVision for designing mobile apps, it would be great if there were an existing database of what users are designing and can provide suggestions for a specific domain. Another participant proposed an add-on for current UI design toolkits that provides a checklist for essential GUI components when designing a screen. A participant also stated that it would be great to have a tool that checks the completeness of a sketch and gives recommendations.

The main drawback mentioned by the study participants is that they wanted more recommendations. As shown in the results section, GUIFETCH was computed recommendations for 65% of the sketch screens, but it could not find any recommendation for the remaining 35% of the screens. Although the availability of the recommendations ultimately depends on which open source apps are available, I manually inspected the cases for which no recommendations were provided. I found that some of these sketch screens were not part of the essential functionality of their corresponding app category. Another reason I found has to do with drawing very ad-hoc screens. An address book sketch drawn by one of the participants, for instance, used so many personal names and images that GUIFETCH was not able to find any match for it. Another limitation, which I plan to address in future work, is that GUIFETCH is currently unable to detect and compare images.

Besides the limitations mentioned by the participants, the initial experience shows that (unsurprisingly, in hindsight) it is easier to find good matches for types of apps that tend to provide a standard set of features, such as address books. Finding good matches for app categories that vary broadly (*e.g.,* fitness apps) or app categories that use many non-conventional GUI widgets (*e.g.,* games) tends to be more challenging. This seems to indicate that the technique may be more suitable for certain kinds of apps than for others.

CHAPTER 5

AUTOMATED TEST MIGRATION FOR MOBILE APPS

This chapter describes APPTESTMIGRATOR, a technique that migrates test cases between apps that are developed based on the same or similar specification or apps that simply belong to the same category. The work described here was originally published in [6, 5]. The rest of this chapter is structured as follows. Section 5.1 illustrates a motivating example, Section 5.2 presents the technique, and Section 5.3 discusses the evaluation of the technique both in the context of education and general apps.

5.1 Motivating Example

In this section, I present a motivating example that consists of two shopping list apps, a source app and a target app, and a test case for the source app (source test). The goal of APPTESTMIGRATOR is to migrate the source test from the source app to the target app. Fig. 5.1 shows the sequence of events and assertions that the test case triggers to check the "sorting items in list" functionality in the source app. (The sequence of events to add items to the list are not shown due to space limitations.) The test clicks on "More options" (a→b), then "Sort..." (b→c), and finally "A - Z" (c→d). It then checks whether the items are displayed in the sorted order by using six assertions. Items that are checked by assertions are highlighted with rectangles on screen *d*.

Fig. 5.2 shows the sequence of events and assertions generated by APPTESTMIGRATOR for the target app after migration. The test clicks on "More options" (a→b), then "Settings" (b→c), and finally on "Default style ordering of items" (c→d). Next, it selects "Ascending Alphabetical Order" (d→e). It then clicks the "Navigate up" button (e→f) and checks whether the items are displayed in the sorted order through six assertions. Also here, items that are checked by assertions are highlighted with rectangles on screen *f*.

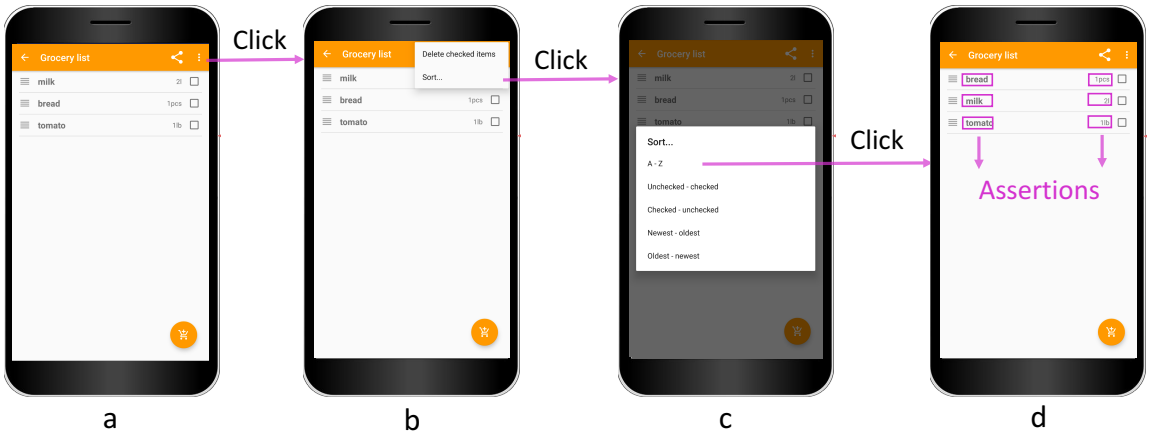

Figure 5.1: Sequence of GUI events of a test case that sorts items in a list in the source app.

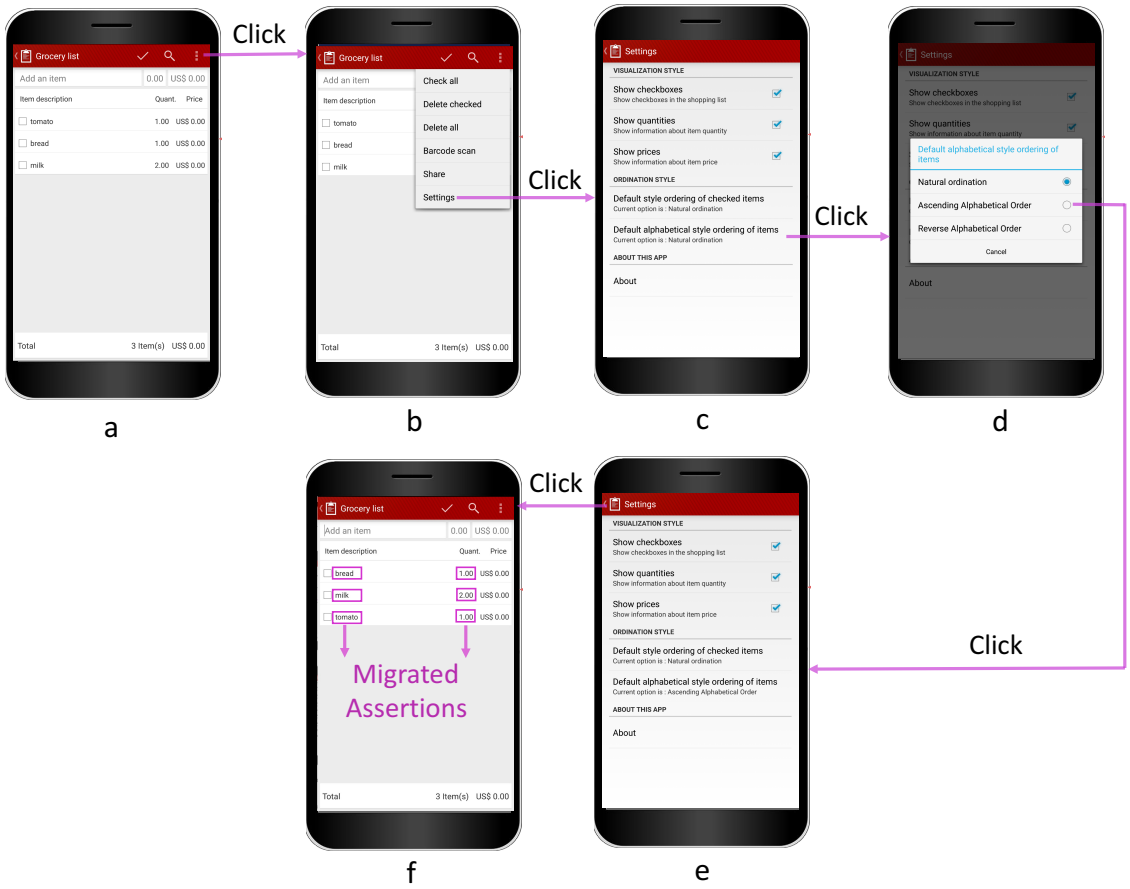

Figure 5.2: Sequence of GUI events migrated by AppTestMigrator from the source app (Fig. 5.1) to the target app.

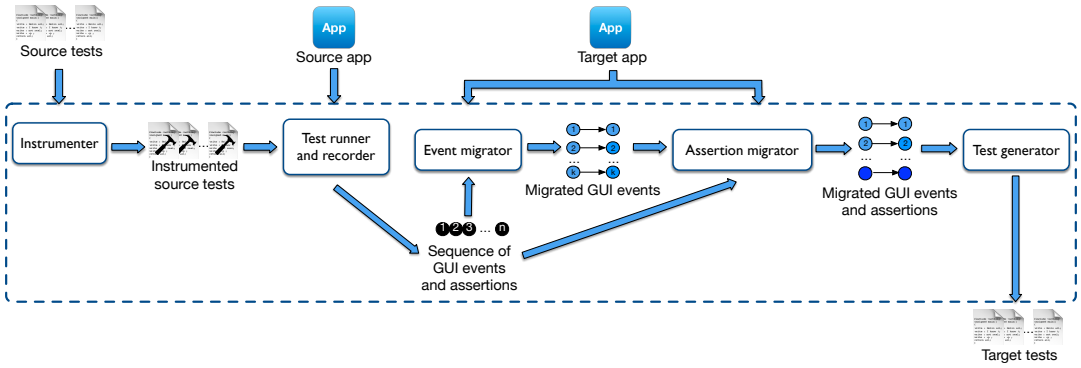

Figure 5.3: Overview of APPTESTMIGRATOR.

As the example shows, although the apps belong to the same category, and provide similar functionality, their GUIs are quite different, which makes it difficult to migrate test cases from the source to the target app.

5.2 Technique

Fig. 4.3 shows an overview of the technique, APPTESTMIGRATOR. As the figure shows, APPTESTMIGRATOR takes as input the *source app*, the *source tests* (a set of tests for the source app), and the *target app*, and produces as output the *target tests* (the source tests migrated to the target app). APPTESTMIGRATOR consists of five main modules: *Instrumenter*, *Test runner and recorder*, *Event migrator*, *Assertion migrator*, and *Test encoder*. First, *Instrumenter* instruments the source tests so that *Test runner and recorder* can record both the sequence of events generated and the assertions (i.e., oracles) checked by the tests. *Event migrator*, and subsequently *Assertion migrator*, then migrate the events and assertions from the source app to the target app. Finally, given the migrated events and assertions, *Test encoder* generates actual test cases for the target app. In the rest of this section, I discuss each step in detail.

5.2.1 Instrumenter

Testing frameworks provide APIs that let users write test cases by generating events and assertions. The *Instrumenter* module instruments these APIs to collect information about GUI interactions.

5.2.2 Test Runner and Recorder

The *Test runner and recorder* module runs the instrumented source tests to record the sequence of events they generate and the assertions they check. For each event, it logs the performed action, the element that is the target of the action, and the input value used by the action, if any.

5.2.3 Event Migrator

The *Event migrator* is one of the key modules of our technique. Given (1) a sequence of source events extracted from a source test and (2) the target app, this module tries to migrate the source events to the target app. Algorithm 1 (MIGRATEEVENTS) describes how this module operates.

The algorithm first computes a *Window Transition Graph (WTG)* [85] for the target app (line #5)—a statically-computed graph where nodes represent windows (i.e., activities, menus, and dialogs) and edges represent transitions between windows, triggered by callbacks executed in the UI thread [14]. The algorithm then launches the target app (line #6).

For each source event, the algorithm tries to find a match in the target app (target event) within a predefined time limit (lines #7-74). To do so, it gets the GUI state and finds all the elements in the GUI state with which users can interact: it first traverses the GUI state and checks the value of the attributes of the elements and of their ancestors; it then identifies *actionable elements*, that is, elements that are clickable, long-clickable, or checkable (lines #14–15).

Algorithm 1 Algorithm for migrating events.

Input: $srcEvents, targetApp$
Output: $targetEvents$

procedure MIGRATEEVENTS
  $MR \leftarrow MAX\_CONSECUTIVE\_RANDOM\_EVENTS$
  $MT \leftarrow MATCH\_THRESHOLD$
  $targetEvents \leftarrow$ List$< tEvent >$
  $wtg \leftarrow$ computeWTG($targetApp$)
  launchTargetApp($targetApp$)
  for $index \leftarrow 1$ to $srcEvents.size()$ **do**
    $currSrcEv \leftarrow srcEvents$.get($index$)
    $matched \leftarrow$ False
    $numRandEvents \leftarrow 0$
    $eventsTriggered \leftarrow$ List$< tEvent >$
    while !timeout() **do**
      $nextEvent \leftarrow$ null
      $state \leftarrow$ getGUIState()
      $actionables \leftarrow$ findActionables($state$)
      $mScore \leftarrow 0$
      for each $actionable$ in $actionables$ **do**
        $score \leftarrow$ computeSimilarityScore($currSrcEv$, $actionable$)
        if $score > MT \wedge score >= mScore$ **then**
          if $nextEvent$!= null **then**
            $nextEvent \leftarrow actionable$
          else
            $nextEvent$.setAlternatives($actionable$)
          end if
          $mScore \leftarrow score$
        end if
      end for
      if $nextEvent$!= null **then**
        $matched \leftarrow$ True
        $eventsTriggered$.add($nextEvent$)
        $targetEvents$.addAll($eventsTriggered$)
        triggerEvent($nextEvent$)
        break
      else
        $sActionables \leftarrow$ findStaticActionables(wtg)
        $mScore \leftarrow 0$
        for EACH sA IN $sActionables$ **do**
          $score \leftarrow$ COMPUTESIMILARITYSCORE($currSrcEv$, sA)
          if $score > MT \wedge score > mScore$ **then**
            $staticNextEvent \leftarrow sA$
            $mScore \leftarrow score$
          end if
        end for
        if $staticNextEvent$!= null **then**
          $staticEventState \leftarrow$ getState($staticNextEvent$)
          $sPath \leftarrow$ findShortestPath($state$, $staticEventState$)
          if $sPath$!= null **then**
            $nextEvent \leftarrow sPath$.getFirstEvent()
          end if
        end if
        if $nextEvent$ == null **then**
          if $numRandEvents$!= MR **then**
            $nextEvent \leftarrow$ pickNextEventSemiRandomly()
            $numRandEvents \leftarrow numRandEvents$ + 1
          else
            $nextEvent \leftarrow$ back
            $numRandEvents \leftarrow 0$
          end if
        end if
        triggerEvent($nextEvent$)
        $eventsTriggered$.add($nextEvent$)
      end if
    end while
    if !$matched$ **then**
      $lastMatchedEv \leftarrow$ getLastMatchedSourceEvent($targetEvents$)
      if alternativeEventExists($lastMatchedEv$) **then**
        $alternativeEvent \leftarrow$ getAlternative($lastMatchedEv$)
        $targetEvents$.replaceLastEvent($alternativeEvent$)
        $index \leftarrow index$ - 1
      end if
      launchTargetApp($targetApp$)
      triggerPreviouslyMigratedEvents($targetEvents$)
    end if
  end for
  return $targetEvents$
end procedure

After identifying the actionable elements, the algorithm computes a similarity score between the source event and these elements (call to method *computeSimilarityScore* on line #18, discussed later in this section.) The actionable element with the highest similarity score is considered a match for the source event, if the score is above a given threshold. If the algorithm finds multiple matches, it selects one of them randomly and records the others as alternative matches (lines #19–26). I discuss how the algorithm might use these alternative matches later in this section. If the algorithm can find at least a match, it adds to the target events the actionable element matched (lines #28–31). It then triggers the matched actionable element and continues by trying to find a match for the next source event (lines #32–33).

Conversely, if the algorithm does not find any matches in the current GUI state, it leverages the WTG it statically computed for the target app to find a matching element elsewhere in the app. Specifically, it finds all the actionable elements in the WTG and tries to match them against the current source event. Also in this case, the actionable element with the highest similarity score (above a given threshold) is considered a match for the source event (lines #35–43). If one such match is found, the algorithm tries to find the shortest path $sPath$ in the WTG from the current GUI state to the GUI state where the matched actionable element exists (lines #44–46). If $sPath$ exists, the algorithm considers the first actionable element of the path as the next event to trigger (lines #47–49). Note that the algorithm only triggers the first actionable element of the path, rather than triggering them all, because the WTG is computed statically and may contain infeasible paths; roughly speaking, triggering the first event is likely to point the search in the right direction even when the whole path $sPath$ is not feasible. After triggering this element, APPTESTMIGRATOR again tries to find the next match dynamically.

If the algorithm does not find any match, either dynamically or statically, it semi-randomly selects one of the actionable elements that was not been previously selected (lines #51–54). By semi-randomly, I mean that APPTESTMIGRATOR takes into account

the structural relationships between the elements when deciding which interactable elements to select. Assume that there are two forms f_a and f_b in a screen, each containing an EditText box and a submit button, and that the previous match resulted in typing an input into an EditText box in form f_b. In this case, APPTESTMIGRATOR's semi-randomly selection would pick the submit button in f_b as the next element because it is semantically related to the previously selected element. To identify structural relationships between elements, APPTESTMIGRATOR leverages their XPath [84] in the GUI hierarchy of the screen, by favoring elements whose XPath shares longer subpaths with the XPath of the previously selected element. If the number of consecutive random events triggered reaches a predefined limit, the algorithm backtracks to the GUI state where the first random event was triggered (lines #55–58).

If the algorithm times out without finding a match for the current source event ($currSrcEv$), it checks whether an alternative match exists for the last matched source event ($lastMatchedEv$). If so, it (1) backtracks by invalidating the last target event and replacing it with one of the alternative matches (lines #64–70), (2) relaunches the target app (line #71), (3) triggers all the previously migrated events, including the selected alternative match (line #72), and (4) tries to find a match for $currSrcEv$ in this new GUI state. (Relaunching the target app and replaying events is necessary because an app execution cannot be easily backtracked.) Conversely, if there are no possible alternative matches for $lastMatchedEv$, the algorithm skips $currSrcEv$ and tries to match the source event that follows $currSrcEv$. Also in this case, the algorithm relaunches the app and replays the previously matched events up to (and including) the one(s) matching $lastMatchedEv$. It does so to undo the possible triggering of target events that were statically or randomly identified while looking for a match for $currSrcEv$.

After processing all source events, the algorithm returns the sequence of successfully migrated target events (line #75).

Computing the Similarity Score Between Two Events To match a source event ev_s to a target event ev_t, APPTESTMIGRATOR computes their similarity score by comparing the GUI elements targeted by the events (e.g., the button in the case of a button click). I refer to these elements as el_s (source element) and el_t (target element). The technique does so, instead of assessing the similarity of the actual events, because it is possible to provide the same functionality by performing different actions on a target element. For instance, it is possible to select an element by clicking, long clicking, or checking such element. Similarly, APPTESTMIGRATOR does not require matching elements to be of the same type (unless the element is of a type that accepts inputs), as different elements might provide the same functionality (e.g., *Button*, *ImageButton*, and *TextView*, in Android).

Given ev_s and ev_t, the technique extracts all the textual information associated with el_s and el_t. First, if an element has a label, or any label exists within a predefined distance from the element, the technique extracts it. Second, it checks whether the developer defined any content description or hint for the element and, if so, it retrieves the values of these attributes as well. Third, it considers the ID of the element, as developers tend to assign meaningful ids to elements. Finally, if the element is an image, the technique retrieves the image filename, which is also often meaningful.

After extracting these pieces of textual information associated with el_s and el_t, the technique first preprocesses them. In particular, it tokenizes each piece of textual information and applies lemmatization [47] to the resulting tokens. The results of this preprocessing are two sets of tokens, one for el_s (set_s) and one for el_t (set_t). The technique then considers all possible pairs of tokens (tok_s, tok_t), such that $tok_s \in set_s$ and $tok_t \in set_t$, and computes two distance scores for each such pair based on (1) edit distance [72] and (2) semantic similarity. To compute this latter, I created an ontology for mobile apps based on word embeddings that I generated using the Word2Vec [54] methodology. Specifically, to build the ontology, I generated a Word2Vec model using 500 randomly selected user manuals for mobile apps in different categories. The user manuals contain sets of instructions that

correspond to different user scenarios for a given app. For each word, the model produces a feature vector, and the semantic distance between two words is determined by the cosine similarity between their vectors. Building this ontology allows us to improve over my previous work [7], where I used general-purpose lexical databases such as WordNet [56].

After computing the scores based on edit and semantic distance between two tokens, the technique considers the maximum of these two values as the similarity score for these tokens. It then matches each source token with the target token that has the highest similarity score (above a given threshold) and computes the overall similarity score by averaging the scores of all the matched tokens in set_s and set_t.

5.2.4 Applying Algorithm 1 to the Example

I now show how algorithm 1 can migrate the source test (Fig. 5.1) to the target app (Fig. 5.2) in the motivating example.

APPTESTMIGRATOR first matches the clicks on the 3-Dots ("More options") menus in the source and target apps (screen *a* in both figures). The next event in the source test is a click on the element with label "Sort..." (screen *b* in Fig. 5.1). Since a corresponding element does not exist in the target app, APPTESTMIGRATOR does not find any matches. Therefore, it uses the WTG of the target app to find a match in another GUI state of the app and compute the shortest path from the current GUI state to the GUI state where the match exists. Due to space limits, I do not show the WTG and the statically matched element here. For the sake of the example, it suffices to say that the first actionable element of the computed shortest path is menu entry "Settings". Therefore, the technique would trigger that menu entry (screen *b* in Fig. 5.2).

APPTESTMIGRATOR then looks for a match for the next source event (click on menu entry "Sort...") by comparing it against the actionable elements of the current GUI state (screen *c* in Fig. 5.2) and finds two possible matches: "Default style ordering of checked items" and "Default alphabetical style ordering of items". (The technique finds these

matches by considering the semantic relation between the words “sort” and “order” when computing the similarity scores between the tokens associated with the source and target elements.) Since the similarity scores for these two possible matches are the same, the technique triggers one of the two actionable elements (e.g., “Default style ordering of checked items”) randomly and records the other as an alternative match. APPTESTMIGRATOR then tries to find a match for the next source event (click on menu entry “A - Z”), but it does not find any match, either statically or dynamically, within the time limit. Since the previous source event (click on “Sort...”) has an alternative match, APPTESTMIGRATOR invalidates the previous match, “Default style ordering of checked items”, and triggers the alternative match, “Default alphabetical style ordering of items” (screen *c* in Fig. 5.2). It then tries again to find a match for “A - Z”, is able to match it to the *CheckedTextView* element with label “Ascending Alphabetical Order” (due to the semantic similarity of the terms “Ascending” and “A - Z”), and triggers the corresponding actionable element (screen *d* in Fig. 5.2). Note that the specialized ontology for mobile apps (described above) is what allows the technique to identify this semantic similarity, which would have not been possible using a general-purpose lexical database.

5.2.5 Assertion Migrator

Assertions are essential parts of test cases. To better understand how assertions are written for GUI-based test cases, I manually inspected a large number of test cases for Android apps on GitHub. I specifically focused on test cases written using the Espresso testing framework [20], which provides various APIs for writing assertions. As I also explain in Section 5.3.1, I chose Espresso because it is one of the major test automation frameworks.

I randomly selected 500 test cases, which contained 884 assertions and classified these assertions. Table 5.1 shows the categories of assertions that I identified, together with the total number and the percentage of assertions for each category. In total, I identified five main categories, where the most common category is *UI-based*, which accounts for 81.2%

Table 5.1: Assertions statistics.

Category	Property (if applicable)	Total	Percentage
UI-based			
	Displayed	430	48.6%
	Text	224	25.3%
	Clickable	22	2.5%
	EffectiveVisibility	11	1.2%
	Hint	10	1.1%
	SpinnerText	8	0.9%
	Checked	6	0.7%
	CompletelyDisplayed	4	0.5%
	Enabled	3	0.3%
	ContentDescription	1	0.1%
		719	81.2%
Hierarchy-based			
	Child	20	2.2%
	isDescendantOfA	14	1.6%
	Parent	13	1.5%
	hasDescendant	13	1.5%
		60	6.8%
DoesNotExist		50	5.7%
Intent-based		44	5%
Others		11	1.2%

of the considered assertions. Assertions in this category check for properties of elements at a specific point of the execution. The list of such properties is also shown in Table 5.1. The second most common category is *Hierarchy-based* (6.8% of the assertions). Assertions in this category check the relationships between two elements. I show different examples, such as child-parent (*Child*) and parent-child (*Parent*) relationships, in Table 5.1. The third and fourth most common categories are *DoesNotExist* and *Intent-based*. *DoesNotExist* assertions check that a specific element does not exist in a given GUI state, whereas *Intent-based* assertions check specific properties of intents—message objects used within Android to request an action from another component, either within the same app or in another app. The rest of the assertions (1.2%) are specific to their corresponding apps and cannot be easily generalized.

In the rest of this section, I discuss how the *Assertion migrator* module migrates assertions that belong to categories *UI-based*, *Hierarchy-based*, and *DoesNotExist*, which account for more than 93% of the assertions I observed. Note that the properties shown in Table 5.1 for each category are only a subset of the properties supported by the tech-

nique. Other examples include *Focus* and *Sibling* for the *UI-based* and *Hierarchy-based* categories, respectively. I decided not to consider the *Intent-based* category for now, as these assertions are used to test the control flow of different components of the apps, rather than their GUIs, and such control flow might be quite different even among similar apps.

Algorithm 2 describes the technique for migrating assertions. An assertion consists of (1) a condition $cond$ and (2) an element el on which $cond$ is checked. Therefore, to migrate an assertion, the algorithm must migrate both el and $cond$. For each source assertion as_s, consisting of a source element el_s and a source condition $cond_s$, the algorithm operates as follow. First, it launches the target app and triggers the migrated events therein (lines #3–4), so that it reaches the GUI state in which it can start migrating the assertion.

To migrate el_s, the algorithm compares all the elements of the current GUI state with el_s and identifies the target element el_t with the highest similarity score (line #10). To do so, function *findMatch* uses an approach analogous to the one I discussed in Section 5.2.3. If a suitable element cannot be found, the algorithm selects an event randomly, triggers it, and tries to find a match in the new GUI state (lines #11–14). The algorithm keeps exploring randomly until it either finds a match (line #16) or reaches a given time limit (line #8). If it is unable to find a match, either directly or through random exploration, the algorithm skips the current assertion (lines #19–20). Otherwise, it continues and tries to migrate $cond_s$ (lines #22–42) based on its category:

UI-based If $cond_s$ checks a *UI-based* property $prop$, the algorithm gets the expected value val for $prop$ from el_t (line #25). Given el_t, $prop$, and val, the algorithm then generates an assertion for the target app (line #26).

Hierarchy-based If $cond_s$ checks a *Hierarchy-based* property, the algorithm first tries to match the element associated with $cond_s$ with an element in the current GUI state, using again function *findMatch* (lines #28–29). For condition *Parent(parentEl)*, for instance, *parentEl* would be the element in the source app that must be matched in the target app.

Algorithm 2 Algorithm for migrating assertions.

Input: $targetEvents, srcAssertions, targetApp$
Output: $targetAssertions$

procedure MIGRATEASSERTIONS
  for each as_s in $srcAssertions$ **do**
    launchTargetApp($targetApp$)
    triggerMigratedEvents($targetEvents$)
    $mAssertion \leftarrow$ null
    $mElement \leftarrow$ null
    $el_s \leftarrow as_s$.getElement()
    while !timeout() **do**
      $state \leftarrow$ getGUIState()
      $el_t \leftarrow$ findMatch($state$, el_s)
      if el_t == null **then**
        $nextEvent \leftarrow$ pickNextEventRandomly()
        triggerNextEvent($nextEvent$)
        continue
      else
        break
      end if
    end while
    if el_t == null **then**
      continue
    else
      $cond_s \leftarrow as_s$.getCondition()
      $prop \leftarrow as_s$.getConditionProperty()
      if $cond_s$ is UI-based **then**
        $val \leftarrow el_t$.getPropertyValue($prop$)
        $mAssertion \leftarrow$ Assertion(el_t, $prop$, val)
      else if $cond_s$ is Hierarchy-based **then**
        $cElement \leftarrow assertion$.getConditionElement()
        $cmElement \leftarrow$ findMatch(getGUIState(), $cElement$)
        if $cmElement$!= null **then**
          $mAssertion \leftarrow$ Assertion(el_t, $prop$, $cmElement$)
        end if
      else if $cond_s$ is DoesNotExist **then**
        $srcEl, srcEv \leftarrow$ findElement(as_s)
        $trgEl \leftarrow srcEv$.getTargetElement()
        $trgSt \leftarrow targetEvents$.get($trgEl$).getState()
        $lastTrgSt \leftarrow targetEvents$.getLast().getState()
        $ne \leftarrow$ findDoesNotExist(el, $trgSt$, $lastTrgSt$)
        if ne != null **then**
          $mAssertion \leftarrow$ Assertion($cond_s$, ne)
        end if
      end if
    end if
    if $mAssertion$!= null **then**
      $targetAssertions$.add($mAssertion$)
    end if
  end for
  return $targetAssertions$
end procedure

If *findMatch* is able to find a match, the algorithm generates an assertion for the target app using the matched element, the property that needs to be checked, and el_t (lines #30–32).

DoesNotExist If $cond_s$ belongs to category *DoesNotExist*, the algorithm must identify and migrate in the source app an element, $srcEl$, that (1) has the properties specified in the assertion (e.g., a specific label), (2) does not exist in the last GUI state reached by the source test, where the assertion is checked, and (3) exists in a previous GUI state. Note that the last condition is included because, in most if not all cases, this kind of tests are used to check that a previously existing element has been successfully removed. To identify $srcEl$, function *findElement* (line #34) examines the GUI states reached by the source test from the beginning of the execution to the point where the assertion is checked. While doing so, function *findElement* also identifies the last GUI state in which $srcEl$ existed and the source event, $srcEv$, that led to that state. The algorithm then identifies the target event that was mapped to $srcEv$ when migrated, $trgEl$, and the GUI state, $trgSt$, resulting from triggering $trgEl$ (lines #35–36). The algorithm also identifies the GUI state, $lastTrgSt$, resulting from triggering the last target element; that is, the state in the target app in which the migrated element should not exist (line #37). To generate the actual assertion, method *findDoesNotExist* (line #38) checks whether there is an element ne in $trgSt$ that does not exist in $lastTrgSt$ and matches $srcEl$; if so, the algorithm generates a corresponding assertion (lines #39–41).

Finally, the algorithm adds all the assertions that it is able to migrate to the list of migrated assertions (lines #44–46) and return them (line #48).

5.2.6 Applying Algorithm 2 to the Example

In this section, I illustrate how algorithm 2 can migrate the assertions in the source test (Fig. 5.1) to the target app (Fig. 5.2) in the motivating example.

APPTESTMIGRATOR first triggers the migrated source events in the target app, thus

reaching Screen *e* in Fig. 5.2. The technique must then identify in this screen the elements checked by the assertions, shown on Screen *d* in Fig. 5.1. Because it is unable to do so, the technique randomly selects an actionable element in the target app. Assume that at some point it selects the "Navigate up" button (e→f), which takes the target app to screen *f* in Fig. 5.2. The technique again tries to find matches for the elements in the new GUI state, and this time it is successful. It then checks the property of the conditions in the assertions, which are all "Displayed" (i.e., all *UI-based*). Therefore, APPTESTMIGRATOR generates the assertions using the matched elements and using "Displayed" as the condition to check.

5.2.7 Test Encoder

Once APPTESTMIGRATOR has processed all source events and assertions, the *Test encoder* generates actual test cases for the target app based on the migrated events and assertions.

5.3 Empirical Evaluation

To evaluate the technique, I implemented APPTESTMIGRATOR and investigated the following research questions:

1. **RQ1:** How accurate is APPTESTMIGRATOR in migrating events from source to target apps?
2. **RQ2:** Is APPTESTMIGRATOR more effective than GUITESTMIGRATOR in migrating test events?
3. **RQ3:** How accurate is APPTESTMIGRATOR in migrating oracles from source to target apps?

5.3.1 Implementation

The implementation supports Android apps, as Android is one of the major platforms in the mobile app market. APPTESTMIGRATOR requires as input tests for the source app,

and the current implementation supports tests written using the Espresso testing framework [20]. I chose Espresso for several reasons, including the fact that it is widely used and is actively maintained by Google, provides easy access to a more complete GUI state, is integrated with Android Studio's Espresso Test Recorder, and supports asynchronous tasks. I modified Espresso to collect relevant dynamic information during test execution. Note, however, that my general approach is not specific to Android and Espresso and could be ported to other mobile platforms and testing frameworks. For the static analysis, I leverage gator [85], a static analysis tool that creates a model of the GUI-related behavior of an Android app. I used Neo4j [57], a graph database management system, to interact with the static model. To generate the Word2Vec model, I used Genism [69], an open-source vector space and topic modeling toolkit implemented in Python.

5.3.2 Evaluation Setup

To evaluate APPTESTMIGRATOR, I first identified app categories in the Google Play Store containing at least four apps with over 1,000 installs and source code available.[1] This resulted in many categories, including address book, diet tracking, expense tracking, food ordering, mail clients, music, news, note taking, online shopping, shopping list, to-dos management, and weather apps. I then excluded those categories that are too broad to provide a standard set of features (e.g., games and fitness). Among the resulting categories, I randomly selected four: *Shopping List*, *Note Taking*, *Expense Tracking*, and *Weather* apps. Finally, for each of these four categories, I selected the four apps either with the most test cases or randomly, in case there were not enough apps with test cases in a category.

In total, I found two apps with GUI test cases: one in the *Shopping List* category (six test cases), and one in the *Note Taking* category (four test cases). I then asked eight CS students not involved in this research but familiar with (when not expert in) testing to write more

[1] Because the current implementation of APPTESTMIGRATOR supports test cases written in Espresso, the app source code is required to build and run test cases. However, the technique could be directly applied to binary apps by targeting a different testing framework.

Table 5.2: Description of the benchmark apps and tests.

ID	Name	LOC	Installs	#Test cases	#Events	#Oracles	Coverage
S1	Shopping List	6.7K	10,000+	11	61	20	67%
S2	Shopping List	8.2K	100,000+	11	44	12	60%
S3	Shopping List	18.2K	5,000+	10	44	16	54%
S4	OI Shopping List	24.9K	1,000,000+	16	71	20	51%
N1	Note Now	7.7K	1,000+	8	35	16	53%
N2	Swiftnotes	5.7K	5,000+	11	56	24	68%
N3	Writeily Pro	9.3K	5,000+	9	49	12	57%
N4	Pocket Note	13.4K	1,000+	10	50	13	50%
E1	EasyBudget	14.1K	50,000+	10	53	11	52%
E2	Expenses	5.2K	1,000+	8	43	10	90%
E3	Daily Budget	7.1K	10,000+	10	49	14	69%
E4	Open Money Tracker	14.5K	1,000+	10	70	27	56%
W1	Forecastie	8.6K	10,000+	9	44	13	53%
W2	Good Weather	11.6K	5,000+	9	40	19	59%
W3	World Weather	18.5K	1,000+	8	60	6	59%
W4	Geometric Weather	37.5K	10,000+	8	28	8	30%

test cases for all the apps using BARISTA [22]—a test record and replay tool that allows users to generate tests in a visual and intuitive way. I first introduced the participants to the BARISTA tool and gave them some time to become familiar with it and ask us questions about it. I then asked each participant to write test cases for two randomly selected apps.

Table 5.2 shows the list of the apps and tests I used. For each app, the table shows its ID, name, size (LOC), number of installations, and number of test cases, in addition to the total number of events, the total number of oracles, and the statement coverage for the test cases considered. Note that, although I considered only open-source apps, these apps are available in the Google Play Store and have been installed at least 1,000 times, as mentioned above, which should provide some confidence in their quality and popularity.

I applied APPTESTMIGRATOR to these apps and test cases by considering each app in a category as the source app and the remaining apps as target apps. I also compared APPTESTMIGRATOR with its most closely related technique, GUITESTMIGRATOR [7] (GTM, for brevity, in the rest of this section). To do so, while reducing the cost of the manual check of the results, I randomly selected half of the possible combinations of source and target apps and ran GTM on them. I then compared the accuracy of migrating events by GTM with that of APPTESTMIGRATOR. Because GTM does not support the migration of oracles, I could not compare that part of the approach.

5.3.3 Results

Table 5.3 shows the results obtained through manual inspection of the migrated test cases by APPTESTMIGRATOR (both events and oracles) and GTM (events only). The results for the combinations of source and target apps not considered (see Section 5.3.2) are indicated with a dash, "–". The first and second columns of the table show the ID of the source and target apps, respectively. Columns 3 (*completely migrated*) and 4 (*partially migrated*) show the percentage of test cases for which the technique generated complete and partial target test cases, respectively. Completely migrated tests are those for which all events are successfully migrated. Partially migrated tests are those that are not completely migrated but for which at least one event is successfully migrated.

The fifth column (*correctly matched*) indicates the percentage of individual events in the source tests that were correctly matched to events in the target app. To get a better understanding of the performance of the techniques, I manually inspected the events (or oracles) that were not successfully matched and classified them in one of three categories: (1) *unmatched (!exist)* events (oracles) are events (oracles) in the source test that could not be matched to a corresponding event (oracle) in the target app because they actually do not have a counterpart in that app (i.e., true negatives); (2) *unmatched (exist)* events (oracles), conversely, represent events (oracles) in the source test that have a counterpart in the target app, but were nevertheless not migrated (i.e., false negatives); finally, *incorrectly matched* events (oracles) are events (oracles) that were mapped to the wrong events (oracles) in the target app (i.e., false positives). Table 5.3 shows the results with respect to this further classification, in Columns 6 to 8.

Note that, initially, I considered measuring APPTESTMIGRATOR's effectiveness also in terms of coverage and fault-detection ability of the migrated tests. I ultimately decided against it because I believe that these metrics make little sense in the context of test migration; the degree of coverage and fault-revealing ability of the migrated tests not only depends on the number and variety of the source tests, but also on the specifics of the code

on which they run, which makes a meaningful computation of these metrics extremely difficult. A migrated test that exercises an important feature that only represents 1% of the source code, for instance, can be very useful but is not going to affect much the overall coverage or fault revealing ability of the migrated test suite. In other words, I believe that these metrics provide little to no information on the effectiveness of the technique itself, which is what I are interested in assessing.

Table 5.3: Results of migrating test cases using APPTESTMIGRATOR (ATM) (both events and oracles) and GTM (events only).

Source app	Target app	Completely migrated		Partially migrated		Correctly matched			Unmatched (!exist)			Unmatched (exist)			Incorrectly matched		
						ATM		GTM	ATM		GTM	ATM		GTM	ATM		
		ATM	GTM	ATM	GTM	Events	Oracles	Events	Events	Oracles	Events	Events	Oracles	Events	Events	Oracles	Events
S1	S2	54%	27%	46%	36%	63%	65%	31%	18%	15%	17%	4%	0%	35%	15%	20%	17%
S1	S3	55%	-	45%	-	65%	65%	-	20%	25%	-	3%	5%	-	12%	5%	-
S1	S4	36%	-	46%	-	30%	30%	-	35%	35%	-	8%	30%	-	27%	5%	-
S2	S1	45%	-	45%	-	68%	84%	-	22%	8%	-	0%	0%	-	10%	8%	-
S2	S3	55%	36%	36%	27%	71%	83%	48%	20%	17%	15%	2%	0%	22%	7%	0%	15%
S2	S4	55%	36%	45%	27%	76%	75%	42%	10%	25%	18%	2%	0%	17%	12%	0%	23%
S3	S1	60%	30%	40%	30%	68%	88%	40%	20%	6%	23%	5%	6%	23%	7%	0%	14%
S3	S2	40%	30%	50%	50%	64%	75%	45%	14%	0%	14%	9%	19%	23%	13%	6%	18%
S3	S4	30%	-	20%	-	28%	50%	-	36%	0%	-	18%	44%	-	18%	6%	-
S4	S1	44%	-	31%	-	49%	50%	-	28%	40%	-	7%	0%	-	16%	10%	-
S4	S2	50%	-	40%	-	47%	45%	-	21%	25%	-	18%	25%	-	14%	5%	-
S4	S3	44%	44%	25%	31%	45%	55%	44%	32%	25%	33%	3%	20%	7%	20%	0%	16%
Avg.	-	47%	34%	39%	34%	56%	64%	42%	23%	18%	20%	7%	12%	21%	14%	6%	17%
N1	N2	50%	-	50%	-	69%	81%	-	17%	19%	-	0%	0%	-	14%	0%	-
N1	N3	63%	50%	37%	37%	74%	50%	57%	9%	44%	9%	0%	0%	11%	17%	6%	23%
N1	N4	62%	38%	25%	37%	51%	50%	44%	11%	38%	8%	6%	0%	23%	32%	12%	25%
N2	N1	82%	-	18%	-	88%	100%	-	9%	0%	-	0%	0%	-	3%	0%	-
N2	N3	64%	-	36%	-	73%	79%	-	7%	13%	-	7%	4%	-	13%	4%	-
N2	N4	73%	55%	18%	45%	77%	88%	52%	9%	0%	9%	2%	4%	27%	12%	8%	12%
N3	N1	67%	-	11%	-	64%	58%	-	18%	8%	-	0%	0%	-	18%	34%	-
N3	N2	33%	0%	33%	0%	55%	42%	14%	23%	42%	23%	4%	8%	40%	18%	8%	23%
N3	N4	33%	33%	44%	44%	52%	17%	52%	23%	8%	23%	2%	17%	2%	23%	58%	23%
N4	N1	50%	40%	30%	30%	53%	84%	49%	30%	8%	32%	0%	0%	11%	17%	8%	8%
N4	N2	10%	-	30%	-	32%	0%	-	36%	8%	-	15%	84%	-	17%	8%	-
N4	N3	40%	-	20%	-	41%	54%	-	38%	16%	-	4%	15%	-	17%	15%	-
Avg.	-	52%	36%	29%	32%	61%	59%	45%	19%	17%	17%	3%	11%	19%	17%	13%	19%
E1	E2	50%	30%	50%	40%	58%	36%	43%	19%	36%	15%	0%	0%	17%	23%	28%	25%
E1	E3	50%	-	50%	-	47%	27%	-	34%	9%	-	4%	36%	-	15%	28%	-
E1	E4	50%	30%	30%	30%	47%	55%	45%	26%	0%	32%	0%	9%	10%	27%	36%	13%
E2	E1	62%	38%	38%	50%	74%	70%	55%	11%	0%	16%	2%	10%	7%	13%	20%	22%
E2	E3	75%	50%	0%	37%	38%	70%	45%	40%	0%	40%	0%	0%	13%	22%	30%	2%
E2	E4	62%	-	25%	-	69%	50%	-	18%	20%	-	0%	0%	-	13%	30%	-
E3	E1	70%	-	0%	-	53%	43%	-	31%	29%	-	8%	7%	-	8%	21%	-
E3	E2	60%	-	10%	-	47%	36%	-	47%	43%	-	0%	0%	-	6%	21%	-
E3	E4	50%	30%	20%	30%	67%	21%	41%	13%	36%	16%	10%	7%	31%	10%	36%	12%
E4	E1	40%	-	0%	-	37%	11%	-	28%	52%	-	16%	26%	-	19%	11%	-
E4	E2	30%	10%	30%	40%	43%	33%	31%	33%	30%	33%	4%	26%	22%	20%	11%	14%
E4	E3	50%	-	30%	-	36%	44%	-	33%	30%	-	13%	26%	-	18%	0%	-
Avg.	-	54%	31%	24%	38%	51%	41%	43%	28%	24%	25%	5%	12%	17%	16%	23%	15%
W1	W2	44%	-	44%	-	55%	15%	-	9%	46%	-	7%	31%	-	29%	8%	-
W1	W3	44%	33%	44%	44%	68%	15%	43%	20%	62%	25%	0%	0%	18%	12%	23%	14%
W1	W4	22%	-	67%	-	36%	0%	-	34%	85%	-	5%	15%	-	25%	0%	-
W2	W1	44%	22%	56%	67%	63%	11%	35%	25%	68%	50%	2%	0%	7%	10%	21%	8%
W2	W3	44%	-	44%	-	52%	16%	-	32%	37%	-	3%	5%	-	13%	42%	-
W2	W4	33%	0%	22%	67%	30%	16%	18%	35%	74%	45%	15%	10%	20%	20%	0%	17%
W3	W1	50%	-	37%	-	47%	33%	-	43%	50%	-	0%	17%	-	10%	0%	-
W3	W2	38%	-	50%	-	30%	17%	-	17%	50%	-	2%	0%	-	51%	33%	-
W3	W4	25%	25%	50%	50%	20%	17%	30%	37%	66%	43%	5%	17%	15%	38%	0%	12%
W4	W1	50%	-	37%	-	64%	50%	-	22%	50%	-	0%	0%	-	14%	0%	-
W4	W2	37%	37%	50%	50%	61%	50%	61%	14%	25%	14%	7%	12%	7%	18%	13%	18%
W4	W3	50%	50%	37%	37%	71%	38%	71%	11%	62%	11%	0%	0%	0%	18%	0%	18%
Avg.	-	40%	28%	45%	52%	50%	23%	43%	25%	56%	31%	4%	9%	11%	21%	12%	15%

5.3.4 RQ1: Accuracy in Migrating Events

As Table 5.3 shows, on average, APPTESTMIGRATOR *completely migrated* 47%, 52%, 54%, and 40% of the tests considered, and *partially migrated* 39%, 29%, 24%, and 45% of the tests considered, for the *Shopping List*, *Note Taking*, *Expense Tracking*, and *Weather* categories, respectively. This corresponds to correctly matching 56%, 61%, 51%, and 50% of the events in the four categories. Among the unmatched events, 23%, 19%, 28%, and 25% were true negatives, 7%, 3%, 5%, and 4% were false negatives, and 14%, 17%, 16%, and 21% were false positives.

In summary, on average, APPTESTMIGRATOR completely migrated 48% and partially migrated 34% of the tests considered, while correctly matching 54% of the events. Of the unmatched 46% events, 24% were true negatives, 5% were false negatives, and 17% were false positives.

5.3.5 RQ2: Comparison with GTM

As Table 5.3 shows, on average, GTM completely migrated 34%, 36%, 31%, and 28% of the tests considered, and partially migrated 34%, 32%, 38%, and 52% of the tests considered, for the *Shopping List*, *Note Taking*, *Expense Tracking*, and *Weather* categories, respectively. This corresponds to correctly matching 42%, 45%, 43%, and 43% of the events in the four categories. Among the unmatched events, 20%, 17%, 25%, and 31% were true negatives, 21%, 19%, 17%, and 11% were false negatives, and 17%, 19%, 15%, and 15% were false positives.

In summary, on average, GTM completely migrated 16% fewer tests than APPTESTMIGRATOR (32% versus 48%) and partially migrated 5% more tests than APPTESTMIGRATOR (39% versus 35%). Also on average, GTM correctly matched 11% less event than APPTESTMIGRATOR (43% versus 54%). Finally, the percentages of true negatives, false negatives, and false positives for GTM are 23%, 17%, and 16%, which are 1% less, 12% more, and 1% less than the corresponding results for APPTESTMIGRATOR.

Table 5.4: Benchmarks assertions statistics.

Category	Property (if applicable)	Total	Percentage
UI-based			
	Text	110	46%
	Displayed	84	35%
	Enabled	16	7%
	CompletelyDisplayed	8	4%
	Checked	5	2%
	Focus	4	1%
Hierarchy-based			
	Child	4	1%
DoesNotExist		4	1%
Others		6	3%

To better understand which aspects of APPTESTMIGRATOR allowed it to outperform GTM in most cases, I manually inspected the cases labeled as *correctly matched* for APPTESTMIGRATOR but not for GTM. I found that the new approach for computing similarity scores, the use of static analysis, and the new crawling algorithm helped APPTESTMIGRATOR to successfully migrate the events for which GTM failed in 41%, 36%, and 23% of the cases, respectively.

5.3.6 RQ3: Accuracy in Migrating Oracles

Table 5.4 shows the properties that are checked by the assertions in the tests I considered in our evaluation. As the table shows, the most checked property is *Text* (46%), followed by *Displayed* (35%), *Enabled* (7%), *CompletelyDisplayed* (4%), *Checked* (2%), *Focus* (1%), *Child* (1%), and *DoesNotExist* (1%). The assertions in category *Others* check whether the elements in an *AdapterView* [1] have specific names using custom APIs that are not currently supported by the implementation.

As the results in Table 5.3 show, APPTESTMIGRATOR correctly matched 64%, 59%, 41%, and 23% of the assertions in the *Shopping List*, *Note Taking*, *Expense Tracking*, and *Weather* categories, respectively. Among the unmatched assertions, 18%, 17%, 24%, and 56% were true negatives, 12%, 11%, 12%, and 9% were false negatives, and 6%, 13%, 23%, and 12% were false positives.

In summary, on average, APPTESTMIGRATOR correctly matched 47% of the assertions in the test cases. Of the remaining 53%, 29% were true negatives (assertions that APPTESTMIGRATOR could not match and that in fact had no counterpart in the target app), 11% were false negatives (assertions that APPTESTMIGRATOR could not match but had a counterpart in the target app), and 13% were false positives (assertions without a counterpart in the target app that APPTESTMIGRATOR matched incorrectly). Whereas fault negatives simply make a migrated test potentially less useful, as developers would have to manually check its results, false positives may result in erroneous assertions and, ultimately, in erroneous test outcomes. This issue could be addressed by using the technique as a recommender system, as it is typical for this kind of automated approaches: APPTESTMIGRATOR would propose to the developers the migrated test cases, so that they would have a chance to check them before use.

5.4 Threats To Validity

The primary threat to the external validity of the results concerns whether they will generalize to other apps, tests, and categories. To mitigate this issue, I used randomly selected real-world apps from four different categories. A first threat to internal validity is that the students who wrote some of the tests used in the evaluation were not familiar with the apps under test. However, it is not uncommon for testers to test software they did not develop themselves. A second threat to internal validity consists of possible mistakes in the manual inspection of the test migration results. Despite the many differences in the GUI design of apps within the same category, however, the checks were time consuming but ultimately straightforward, due to the similarity in the functionality behind the GUIs.

CHAPTER 6

LEVERAGING EXECUTION TRACES TO GENERATE TEST INPUTS

This chapter describes GUITESTGEN, a technique that leverages the execution traces of apps that belong to the same category that are generated by their end-users to generate GUI test inputs for other apps in that same category (i.e., apps that share part of their functionality). Unlike APPTESTMIGRATOR that tries to migrate individual tests one by one, GUITESTGEN takes advantage of the combination of multiple execution traces that correspond to the same functionality to synthesize tests for that functionality. In the rest of this chapter, I present the technique for GUITESTGEN and discuss the evaluation of the technique.

6.1 Technique

Figure 6.1 shows an overview of the technique. As the figure shows, the technique consists of six main modules: *Instrumenter*, *Recorder*, *Splitter*, *Clusterer*, *FSM generator*, and *Test generator*. First, *Instrumenter* instruments the training apps that belong to the same category so that when users use the apps *Recorder* can record the execution traces. *Splitter* then tries to split the traces in a way that each subtrace ideally represents a functionality. Given the splitted execution traces, *Clusterer* then tries to cluster the subtraces so that each cluster contain those subtraces that represent the same functionality. Next, *FSM generator* generates a finite state machine (FSM) for each cluster. Finally, *Test generator* uses the generated FSMs to guide the exploration of the target app (another app that belongs to the same category as training apps) and generate test inputs for the app. In the rest of this section, I discuss each step in more details.

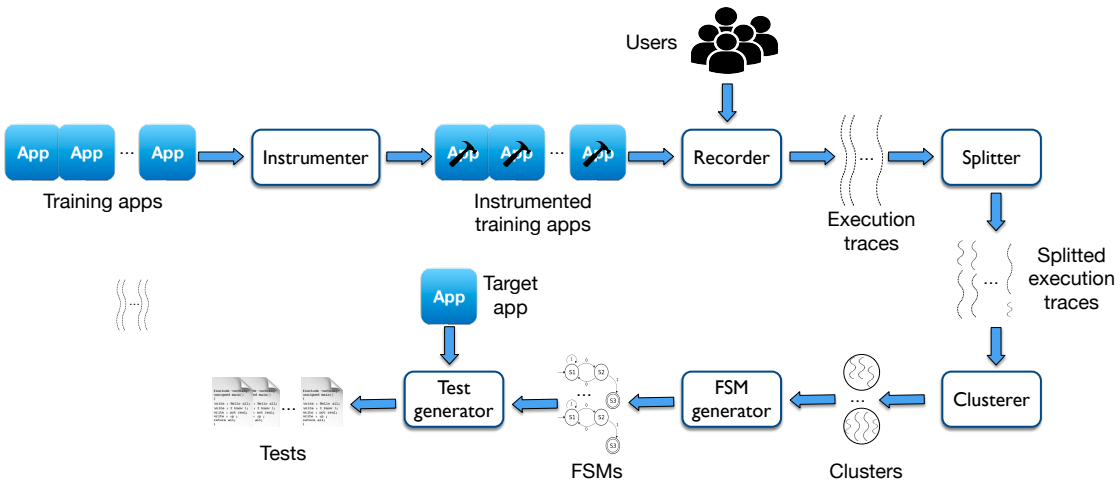

Figure 6.1: Overview.

6.1.1 Instrumenter

Instrumenter instruments the training apps to capture different execution properties that characterize user interactions in the apps. Specifically, it captures call stacks to identify method calls and their depth. For each method call, *Instrumenter* extracts package, class, and method names associated with the call. It also records the names of the activities and fragments explored during the execution. For each activity/fragment, it records starting point and ending point. It also records user events, where a user event can be either a touch event or a keyboard event. A touch event is associated with a widget (e.g., a button), for which *Instrumenter* records, if available, id, associated text, and content description. A keyboard event is associated with a key label. This information is stored in an execution trace and is used in later steps to split the execution trace.

6.1.2 Recorder

Recorder simply records the execution traces generated while users use the instrumented training apps.

6.1.3 Splitter

To split the traces into subtraces, *Splitter* uses *FeatureFinder* [65], a technique that uses a bottom-up approach to identify features by first splitting traces into unit segments, then using a classifier-based algorithm to cluster consecutive, related unit segments, and finally labeling the clustered segments as features that represent different functionalities.

6.1.4 Clusterer

Given the splitted execution traces, *Clusterer* is responsible to cluster the subtraces in a way that each cluster contains those subtraces that represent the same functionality. Each subtrace is a sequence of GUI events (or simply events). To cluster the subtraces, *Clusterer* first extracts the textual information associated with the events using the approach mentioned in Chapter 5, Section 5.2.3. It then represents each subtrace as a sequence of words where each word is the textual information associated with an event. It then performs semantic embedding using Doc2Vec [39], which generates a vector for each subtrace. Finally, it uses DBSCAN [21] to cluster the subtraces.

6.1.5 FSM generator

Once subtraces are clustered, *FSM generator* is responsible to generate an FSM that represents the subtraces in that cluster. In each FSM, states represent the target of the actions that are associated with the events and transitions represent the actions that are associated with the events.

To generate an FSM, *FSM generator* first finds an initial event for each cluster. Algorithm 3 describes the process to find an initial event for a cluster. To find an initial event, *FSM generator* compares the initial event in each subtrace (*initialEvent*) with events in other subtraces (*event*) using method *computeSimilarityScore*, discussed in Chapter 5, Section 5.2.3 (lines #7-17). For each subtrace, it then records the number of similar events in other subtraces along with the average of the position of these similar events (lines #18-

Algorithm 3 Algorithm for finding an initial event in a cluster.

```
1:  Input: cluster
2:  Output: initialEvent
3:  findInitialState()
4:  initialEvent ← null
5:  MT ← MATCH_THRESHOLD
6:  similarEvents ← List << Subtrace, Integer, Double >
7:  for each subtrace in getAllSubtraces(cluster) do
8:      numSimilarEvents ← 0
9:      avgIndex ← 0
10:     initialEvent ← subtrace.get(0)
11:     otherSubtraces ← getAllSubtracesExcept(cluster, subtrace)
12:     for each otherSubtrace in otherSubtraces do
13:         mScore ← 0
14:         foundSimilarEvent ← False
15:         for each event in otherSubtrace.getEvents() do
16:             score ← computeSimilarityScore(event, initialEvent)
17:             if score > MT ∧ score >= mScore then
18:                 mScore ← score
19:                 avgIndex ← avg(avgIndex, events.getIndex(event))
20:                 foundSimilarEvent ← True
21:             end if
22:         end for
23:         if foundSimilarEvent then
24:             numSimilarEvents ← numSimilarEvents + 1
25:         end if
26:     end for
27:     similarEvents.add(subtrace, numSimilarEvents, avgIndex)
28: end for
29: ranking ← rankEvents(similarEvents)
30: topRank ← ranking.getTopRank()
31: if topRank.size() > 1 then
32:     initialEvent ← pickRandom(topRank)
33: else if topRank.size() == 1 then
34:     initialEvent ← topRank.get(0)
35: end if
36: return initialEvent
```

28). Finally, it ranks the initial events based on two criteria: the number of similar events and the average of the position of the similar events in their corresponding subtraces (lines #29). In other words, the initial events with the higher number of similar events and the lower average of the position of the similar events are ranked higher. The former criteria has priority over the latter criteria. If the number of top rank events is more than one, one of them is picked randomly. Otherwise, the top rank event is considered as an intial event for the cluster (lines #30-36).

Algorithm 4 next describes the technique to generate an FSM for a cluster. First *FSM generator* finds an initial event for the cluster using Method *findInitialEvent* described earlier (line #5). If the initial event exists, it extracts the target action and the action corresponding to the event as a state and a transition (lines #7-8). It then addes the state to the

FSM (line #9). Next, *FSM generator* modifies each subtrace in the cluster in a way that the event identified as the initial event is the first event in each subtrace and the earlier events in the subtrace are all ignored. If any of the subtraces do not contain the initial event, that subtrace is skipped (lines #10-13). *FSM generator* then continues building an FSM for each cluster. To do so, for each event in each subtrace of the cluster it checks whether the state that could represent the event already exists in the FSM. In other words, *FSM generator* compares the target actions corresponding to the event and the state using the similarity metric discussed in Chapter 5, Section 5.2.3. If it does not find such a state and a transition, it creates a corresponding state and a transition for that event and adds them to the FSM. *FSM generator* continues doing so until all the subtraces are processed (lines #14-22).

Algorithm 4 Algorithm for generating FSMs.

```
 1: Input: cluster
 2: Output: fsm
 3: generateFSM()
 4: fsm ← new FSM()
 5: initialEvent ← findInitialEvent(cluster)
 6: if initialEvent != null then
 7:     from ← initialEvent.getTargetAction()
 8:     transition ← initialEvent.getAction()
 9:     fsm.addState(from)
10:     for each subtrace in getAllSubtraces(cluster) do
11:         if subtrace.contains(initialEvent) != null then
12:             events ← subtrace.getEvents()
13:             events.subEvents(initialEvent, events.getLastEvent())
14:             for each event in events do
15:                 to ← event.getTargetAction()
16:                 if fsm.getState(to) == null then
17:                     fsm.addState(to)
18:                 end if
19:                 fsm.addTransition(from, to, transition)
20:                 from ← to
21:                 transition ← from.getTransition()
22:             end for
23:         end if
24:     end for
25: end if
26: return fsm
```

6.1.6 Test generator

After generating an FSM for each cluster, *Test generator* uses the FSMs to guide the random test generator for the target app. Algorithm 5 describes this step.

Test generator first launches the app under test (line #4). It then tries to use the FSMs

Algorithm 5 Algorithm for generating tests.

```
1:  Input: fsms, testApp
2:  Output: testInputs
3:  generateTests()
4:  launchApp(testApp)
5:  activeFSM ← null
6:  while !timeout() do
7:      state ← getCurrentGUIState()
8:      actionables ← findActionables(state)
9:      for each fsm in fsms do
10:         matchedEvent ← findMatch(actionables, fsm.getInitialState())
11:         if matchedEvent != null then
12:             activeFSM ← fsm
13:             testInputs.add(matchedEvent)
14:             triggerEvent(matchedEvent)
15:             break
16:         end if
17:     end for
18:     if activeFSM == null then
19:         randomEvent ← pickEventRandomly(state)
20:         testInputs.add(randomEvent)
21:         triggerEvent(randomEvent)
22:     else
23:         while activeFSM != null do
24:             state ← getCurrentGUIState()
25:             actionables ← findActionables(state)
26:             states ← activeFSM.getPotentialNextStates()
27:             matchedEvent ← findMatch(actionables, states)
28:             if matchedEvent != null then
29:                 testInputs.add(matchedEvent)
30:                 triggerEvent(matchedEvent)
31:             else
32:                 activeFSM ← null
33:             end if
34:         end while
35:     end if
36: end while
37: return testInputs
```

to generate test inputs in a predefined time threshold (lines #6-37). To do so, it first gets the GUI state and then finds all the GUI elements in the GUI state, which users can interact with them. To find these elements, it traverses the GUI state and checks the value of the attributes of the elements and their ancestors. It then marks those elements that are clickable, long-clickable, or checkable (lines #7-8). I define these elements as "actionable" elements. Once *Test generator* finds the actionable elements in the current GUI state, it checks whether any of the actionable elements matches an intial state in any of the FSMs by computing the similarity score between them using the approach mentioned in Chapter 5, Section 5.2.3. If it is able to find any matches, it considers that FSM as an active FSM, adds that actionable element to the list of test inputs, and triggers the element (lines #9-17). If *Test generator* is not able to find an active FSM, it picks an element in the current GUI

state randomly, adds that element to the list of test inputs, and triggers the element (lines #18-21). Otherwise, it checks whether any of the actionable elements in the current GUI state matches any of the next states in the active FSM. If so, it adds that actionable element to the list of test inputs, and triggers the element. Otherwise, it invalidates the active FSM (lines #23-34). It continues doing this process until a predefined time threshold reaches. Finally, the technique returns the test inputs (line #37).

6.2 Evaluation

To evaluate our approach, we implemented GUITESTGEN and investigated the following research questions:

1. **RQ1:** How does GUITESTGEN compare to random testing in terms of code coverage?
2. **RQ2:** How does GUITESTGEN compare to random testing in terms of finding crashes?
3. **RQ4:** How does GUITESTGEN compare to random testing in terms of covered usage scenarios?

6.2.1 Evaluation Setup

To evaluate GUITESTGEN, I implemented a prototype for GUITESTGEN that supports Android apps. I then used the prototype to conduct a case study on two app categories: *Email Client* and *Music Player*. For each app category, I selected the first four open-source apps that I found to be also available in the Google Play Store. I then instrumented each app using *FeatureFinder* [65] and exercised as many features of the instrumented app as possible, which generated an execution trace for each considered app. Finally, I applied GUITESTGEN to these apps by considering each app in a category as the target app and the remaining apps as training apps. Table 6.1 shows the considered apps. I also ran MONKEY for 10 minutes on each benchmark application, as previous work [12] showed that the test input generation tools for Android apps often hit their maximum coverage

Table 6.1: Benchmark apps.

ID	Training apps	Target app
E1	K-9 MAIL, Tutanota, FairEmail	Criptext
E2	K-9 MAIL, Tutanota, Criptext	FairEmail
E3	Tutanota, FairEmail, Criptext	K-9 MAIL
E4	K-9 MAIL, FairEmail, Criptext	Tutanota
M1	Jockey, Rey-MusicPlayer, Phonograph	VinylMusicPlayer
M2	Jockey, Rey-MusicPlayer, VinylMusicPlayer	Phonograph
M3	Jockey, Phonograph, VinylMusicPlayer	Rey-MusicPlayer
M4	Rey-MusicPlayer, Phonograph, VinylMusicPlayer	Jockey

within few minutes (between 5 and 10 minutes). To gather the crashes in the app, I also collected the entire system log, from the emulator running the app under test. From these logs, I extracted crashes that occurred while the app was being tested. Finally, to measure the covered usage scenarios, I recorded the execution of the tests and manually identified the usage scenarios that were covered by each app.

6.2.2 Results

Table 6.2 shows the results achieved by GUITESTGEN and MONKEY for each of the target apps. The first column shows the ID of the combination of training and target apps. The second and the third columns show the statement coverage achieved by GUITESTGEN and MONKEY, respectively. Column 4 shows the percentage of coverage difference between GUITESTGEN and MONKEY. The fifth and the sixth columns show the number of crashes that occurred while the target app was being tested by GUITESTGEN and MONKEY, respectively. Column 7 shows the number of crashes that was shared between GUITESTGEN and MONKEY. Finally, columns 8 to 10 show the number of usage scenarios covered by GUITESTGEN and MONKEY and the number of shared usage scenarios between them, respectively. Note that in the case of *E4*, I was not able to collect information regarding the number of crashes and the usage scenarios. The reason is that I collected the coverage information earlier than all other information, and the remote server to which the app connected while running was not accessible when I performed the collection of the other

information. I use a dash, ”–” to indicate that this information is missing in Table 6.2.

Table 6.2: Results.

ID	GUITestGen coverage	Monkey coverage	Coverage diff	# GUITestGen crashes	# Monkey crashes	# Shared crashes	# GUITestGen scenarios	# Monkey scenarios	# Shared scenarios
E1	33%	31%	27%	5	1	0	9	6	2
E2	35%	30%	17%	6	7	2	16	17	11
E3	44%	34%	18%	0	7	0	14	13	6
E4	29%	29%	0%	-	-	-	-	-	-
M1	65%	66%	13%	9	9	4	18	15	8
M2	63%	66%	8%	2	2	1	21	16	9
M3	51%	52%	19%	5	5	1	19	20	11
M4	55%	57%	15%	16	3	0	20	15	10

6.2.3 RQ1: Code Coverage

As Table 6.2 shows, on average, GUITestGen and Monkey achieve 35.25% and 31% coverage in *Email Client* category and 58.5% and 60.25% coverage in *Music Player* category, respectively.

For three out of four apps in the *Email Client* category, GUITestGen achieves more coverage than Monkey and for one app both GUITestGen and Monkey achieve the same coverage. On average, the coverage difference between GUITestGen and Monkey in this category is 15.5%.

In the *Music Player* category, Monkey achieves more coverage than GUITestGen for all the four apps. On average, the coverage difference between GUITestGen and Monkey in this category is 13.75%.

6.2.4 RQ2: Crashes

As Table 6.2 shows, the total number of crashes in both categories is 43 and 34 for GUITestGen and Monkey, respectively. Among these crashes, 8 of them are shared between GUITestGen and Monkey.

The number of crashes in the *Email Client* category is 11 and 15 for GUITestGen and Monkey, respectively. Among these crashes, only two of them are shared in this category.

Table 6.3: Distribution of the detected crashes.

Exception Type	Total	GUITESTGEN		Monkey	
		Music player	Email client	Music player	Email client
IllegalStateException	10	3	1	5	1
NullPointerException	7	2	1	4	0
FileNotFoundException	7	5	0	2	0
IllegalArgumentException	6	3	2	1	0
ClassCastException	5	2	0	1	2
IOException	5	2	1	1	1
InterruptedException	4	1	1	1	1
NumberFormatException	2	2	0	0	0
IndexOutOfBoundsException	2	1	0	1	0
UnknownHostException	2	0	0	0	2
StoreClosedException	2	0	1	0	1
SQLiteConstraintException	2	1	1	0	0
ActivityNotFoundException	2	1	0	1	0
MessageRemovedException	2	0	1	0	1
CancellationException	2	1	0	1	0
FolderClosedException	2	0	1	0	1
AddressException	2	0	0	0	2
NameNotFoundException	1	1	0	0	0
RemoteException	1	1	0	0	0
NoSuchMethodException	1	1	0	0	0
TimeoutException	1	1	0	0	0
OperationException	1	1	0	0	0
SSLException	1	0	0	0	1
FieldDataInvalidException	1	1	0	0	0
RuntimeException	1	1	0	0	0
DatabaseObjectNotClosedException	1	0	0	0	1
ParseException	1	0	0	0	1
UnsupportedOperationException	1	0	1	0	0
DeadObjectException	1	0	0	1	0
Exception	1	1	0	0	0

The number of crashes is 32 and 19 in the *Music Player* category for GUITESTGEN and MONKEY, respectively. Among these crashes, 6 of them are shared in this category.

Table 6.3 also shows the distribution of the detected crashes for GUITESTGEN and MONKEY in both *Music Player* and *Email Client* categories.

6.2.5 RQ3: Usage scenarios

As Table 6.2 shows, the total number of usage scenarios in both categories is 117 and 102 for GUITESTGEN and MONKEY, respectively. Among these crashes, 57 of them are shared between GUITESTGEN and MONKEY.

Table 6.4: Usage scenarios for *Email Client* category.

ID	GUITESTGEN scenarios	MONKEY scenarios	Shared scenarios
E1	reply reply all mark as unread filter search star add label	delete backup mailbox turn on signature mark as read	create draft read email
E2	send discard draft hide show mark unread	change view check logs star check legends check spam check trash	change font size delete filter sort create draft check folders search set importance check email encrypt check settings
E3	select reply all send again read receipt sort search clear local messages archive	select all check drafts check spam check starred discard draft show headers check archive	check email mark unread star show folders check sent create draft

In the *Email Client* category, the number of usage scenarios is 39 and 36 for GUITESTGEN and MONKEY, respectively. 19 of these usage scenarios is shared between GUITESTGEN and MONKEY.

In the *Music Player* category, the number of usage scenarios is 78 and 66 for GUITESTGEN and MONKEY, respectively. 38 of these usage scenarios is shared between GUITESTGEN and MONKEY.

To get a better understanding of the differences between the results of GUITESTGEN and MONKEY, I also show the covered usage scenarios by GUITESTGEN and MONKEY in *Email Client* and *Music Player* categories in Table 6.4 and Table 6.5, respectively.

Table 6.5: Usage scenarios for *Music Player* category.

ID	GUITESTGEN scenarios	MONKEY scenarios	Shared scenarios
M1	add to playing queue check song details scan check about change theme remove from playing queue clear playing queue create playlist search change shortcuts	check artist check playlist clear playing queue check genres change library categories change blacklist remember last tab	play/pause music shuffle forward/backward add/remove to/from favorites check albums repeat check folders change color
M2	remove playing queue save playing queue check artist details resert artist image changed colored footers check genres check genre details delete song scan media check about change theme change color	add to playing queue change colored navigation bar change remember last tab change library categories change blacklists Add/remove to/from favorites repeat	check artists play/pause music clear playing queue check playlists shuffle search check albums forward/backward change tag editor
M3	clear playlist check artists clear queue add to queue edit tags create playlist add to playlist play all	enable/disable equalizer save current position check playlist check contact us check licenses change default browser arrange tabs check genres rebuild music library	play/pause music shuffle repeat song forward/backward A-B repeat check directories check about check Now Playing check albums set timer search
M4	create playlist clear queue queue last add to playlist enable/disable equalizer check recently added add include folders remove include folders change default page change background color	check playlists check genres browse set sleep timer remove song	search check songs shuffle play/pause music forward/backward check artists check albums check about repeat show folder

6.2.6 Discussion

The results show that, although GUITESTGEN does not provide dramatic improvements in terms of absolute coverage numbers with respect to MONKEY, it nevertheless provides several advantages when used in addition to MONKEY or other similar techniques. First, GUITESTGEN detects considerably more crashes than MONKEY (an increase of over 25% on average). Second, it is able to cover, exercise scenarios, and trigger crashes that are not covered, exercised, and triggered by MONKEY.

These results suggest that GUITESTGEN, by leveraging existing execution traces collected for real executions, provides a way to incorporate human intelligence into the test-input-generation process. By doing so, our technique can nicely complement other existing automated test-input-generation techniques, such as random testing, and can be combined with them to generate more thorough test sets. The results presented in this chapter also confirm the findings, presented in previous and related work (e.g., [48, 19, 43]), that mining app-execution and app-usage data can help bridge the gap between automated test-input generation and manual test-input generation.

CHAPTER 7
RELATED WORK

In this section, I will focus on works in the areas that are most closely related to my work in this dissertation, which includes GUI builders, generating GUI code from sketches, code search, GUI test repair, and GUI test generation.

7.1 GUI builders

Modern IDEs, such as Eclipse, Xcode, and Android Studio, provide users with GUI builders where users can drag and drop widgets and set the various properties of these widgets. Then, the corresponding code for the GUI is generated. Although this approach might simplify the process for developers, it does not provide enough flexibility to the users during early stages of GUI design, when the users need the freedom to sketch their ideas quickly. Besides, generating GUIs often happens as part of an iterative development approach, where GUIs are constantly updated or changed (*e.g.,* in agile development [23]). Also, the GUI code created by an actual developer is typically higher quality than the code created by an automated tool [87].

7.2 Generating GUI Code from Sketches

GUI sketches are employed in many companies during early stages of user interface design [38]. Some tools allow users to draw sketches and are able to recognize the GUI elements. JavaSketchIt [9] allows creating user interfaces through hand-drawn geometric shapes, identified by a gesture recognizer. SILK [37] allows designers to sketch a GUI using an electronic pad and stylus quickly; it then recognizes widgets and other interface elements as the designer draws them. de Sá and colleagues propose another prototyping

tool that allows users to take a picture of a sketch and is then able to map the sketch elements to mobile widgets [74].

Other tools push this approach even further and not only recognize the GUI elements, but also generate code for the GUIs. MobiDev [76] provides users with a visual language of standard GUI elements that the users can use to draw app sketches; it then generates the apps based on these sketches. REMAUI [58] is another tool that infers GUI code for a mobile app from screenshots or conceptual drawings by using OCR and computer-vision techniques.

Although these tools can generate GUI code from sketches, they have limitations. MobiDev [76] requires users to use predefined GUI elements. REMAUI [58] only supports the top three Android widgets and is limited in generating code for a single GUI and does not support apps with multiple screens and transitions. Moreover, as I also stated above, the code generated automatically might not be as usable and reliable as code written and tested by users.

The closest work is that of Reiss [71], which focuses on searching Java-based GUI code with a single screen. That technique is not directly applicable to mobile apps; creating a precise GUI model for apps is not straightforward because, as I explained in the Introduction, there are multiple ways to build app GUIs (*i.e.,* using XML layout resources, programmatically, or both). Therefore, building a precise model of an app GUI involves analyzing the app using a combination of static and dynamic analyses. Moreover, apps might consist of different screens and transitions. Therefore, it is essential to match the transitions between screens in addition to matching the screens themselves.

7.3 Code Search

Besides commercial code-search engines, such as GitHub (`www.github.com`) and Open-Hub (`code.openhub.net`), there has been a large body of research on code search for helping developers find code in large public open source code repositories (*e.g.,* [2, 4, 10,

11, 18, 63, 29, 27, 33, 32, 34, 40, 45, 46, 55, 62, 73, 79, 80, 81, 83, 82, 86]). In my work on generating GUI code from sketches, I am particularly interested in searching Android apps and validating the GUIs of the apps against the user provided sketches. None of the existing code-search techniques could be directly used for this task, as they require code snippets or a single GUI as the starting point, rather than the sketch of an app.

7.4 GUI Test Repair

Memon [51] proposes a technique that automatically repairs GUI test suites for regression testing. GUITAR [52] repairs test cases that have become unusable due to changes in a GUIs. SITAR [25] is another technique that repairs GUI test scripts written for an earlier version of the same software by using annotated event-flow graphs, a set of transformations, and human input. Grechanik and colleagues propose a GUI test repair technique that automatically identifies changes between GUI objects and uses this information to help testers perform manual repair [26]. ATOM [42] automatically maintains GUI test scripts for evolving mobile applications. Daniel and colleagues propose an approach in which GUI changes are recorded and later used to repair test cases [16]. Huang and colleagues use genetic algorithms to both repair broken GUI test cases and generate new tests [31]. TIGOR is a tool that uses static analysis to help testers determine type errors in GUI scripts [24]. Zhang and colleagues present a technique that uses static and dynamic analysis, along with random testing, to automatically repair broken workflows for evolving GUI applications [88]. The technique by Li and colleagues helps GUI editors map source code to GUI views when code changes [41]. Many researchers propose various approaches to support migration of legacy GUI tests to modern GUI systems by reverse engineering user interfaces (*e.g.,* [17, 44, 78]). WATER uses differential testing to suggest repairs for broken test scripts for web applications [13]. Finally, Omari and colleagues propose a technique for generating XPath extraction queries for a family of websites that contain the same type of information [60, 59]. This work differs from these approaches because it migrates test cases between apps,

rather than repairing during software evolution. In fact, these techniques could not be readily applied in the context of this work, whereas my technique could be used for GUI test repair as well. Another difference between my technique and some of these techniques is that GUITESTGEN is fully automated and does not require any human effort.

7.5 GUI Test Generation

Recently, there have been some works that focus on exploiting the common functionalities between apps to generate more effective GUI tests. Polariz [48] generates test scripts from crowd-based testing by extracting cross-app reusable higher-level event sequences. Augusto [49] generates semantic UI tests based on popular functionalities. Rau, Hotzkow, and Zeller [68, 67] also try to generate efficient GUI tests by learning from tests of other apps. AppFlow [30] leverages machine learning to automatically recognize common screens and widgets to synthesize reusable GUI Tests. Ermuth and Pradel [19] propose a UI-level test generation approach that exploits execution traces of human users to automatically create complex sequences of events. None of the approaches mentioned above focuses on migrating human written test cases with oracles between apps that belong to the same category.

CHAPTER 8
CONCLUSION

Researchers have been mining online repositories for over a decade to take advantage of existing source code to support different development activities. Despite researchers have proposed a number of techniques that leverage existing source code, these techniques mostly focus on supporting coding and maintenance activities and they ignore other important software engineering tasks, such as software design and testing, that have been to a large extent neglected by previous work. To address this limitation, in this dissertation, I defined GUIFETCH, APPTESTMIGRATOR, and GUITESTGEN, three automated techniques that leverage existing source code, test cases, and execution traces to support the design, development, and testing of mobile apps.

GUIFETCH is a code-search technique that takes advantage of the growing number of open source mobile apps in public repositories to provide users with code that can be used as a starting point for the apps they want to create. Given a sketch of an app (i.e., app's screens and transitions between them), GUIFETCH searches for apps in public repositories that are as similar as possible to the provided sketch. The matching apps are then reported to the user, ranked by similarity to the sketch. GUIFETCH can provide developers with a starting point for building their GUI-based apps, support early prototyping, and help designers assess whether there are existing apps similar to the one they want to develop.

APPTESTMIGRATOR is a test migration technique that considers similarities between apps and migrates test cases across similar apps. APPTESTMIGRATOR is motivated by the observation that, although GUIs for different apps can differ dramatically, there are many cases in which apps share similarities that result in conceptually similar GUIs. AppTestMigrator takes as input a source app, a test for the source app (source test), and a target app, and produces as output the source test migrated to the target app (target test). To do so, it

(1) records both the sequence of GUI events generated and the assertions checked by the source test, (2) migrates GUI events and assertions from the source app to the target app using a similarity metric based on a combination of techniques, and (3) generates a target test case based on the migrated events and assertions.

GUITESTGEN is a technique that leverages the execution traces of apps that belong to the same category that are generated by their end-users to generate GUI test inputs for other apps in that same category (i.e., apps that share part of their functionality). This technique is different from APPTESTMIGRATOR that tries to migrate individual tests one by one as GUITESTGEN takes advantage of the combination of multiple execution traces that correspond to the same functionality to synthesize tests for that functionality. To do so, it (1) instruments the training apps, (2) records a set of execution traces for the training apps, (2) splits and clusters the traces, (3) generates a finite state machine (FSM) for each cluster, and (4) uses the generated FSMs to guide the exploration of the test app and generates test inputs for the app.

8.1 Future Work

For the techniques presented in this dissertation, I envision two main research directions for future work.

8.1.1 Supporting App Design and Development

The current version of GUIFETCH focuses on code search for GUI of mobile apps. One possible direction is to extend the technique so that it can also match non-GUI code, rank it, and report it to the users together with the GUI code. Specifically, such a technique could allow users to specify input-output pairs and will add to GUIFETCH the ability to generate glue code between non-GUI and GUI code.

8.1.2 Reusing existing software artifacts within and across programs with similar functionality

Reusing and adapting existing code is one of the common practices in Software Engineering, due to the fact that programs provide similar functionality. When the functionality of programs overlap to a large extent, it is highly likely that the software artifacts for the programs overlap as well. Motivated by this intuition, in my research, I reduced the cost of testing mobile apps by considering similarities between GUIs of apps and automatically migrating test cases across similar apps. Besides testing, other tasks might also benefit from exploiting similar functionality within and across programs. Large programs often contain components that implement similar functionality. Also, some programs might implement similar functionality as other programs. In both of these cases, shared components may share similar vulnerabilities. One possible direction for future work is to investigate how to develop vulnerability detection techniques that are able to detect similar vulnerabilities across similar components by synthesizing new inputs based on information about exploits for one of the shared components. Another research direction could involve investigating how to leverage existing documentation of a program to generate documentation for other programs that share similar functionality, using program analysis and natural language processing techniques.

REFERENCES

[1] *AdapterView*, `https://developer.android.com/reference/android/widget/AdapterView`, 2020.

[2] M. Akhin, N. Tillmann, M. Fahndrich, J. de Halleux, and M. Moskal, "Search by example in TouchDevelop: Code search made easy," in *2012 4th International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE)*, 2012.

[3] M. J. Atallah and S. Fox, *Algorithms and Theory of Computation Handbook*. CRC Press, Inc., 1998.

[4] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: A Search Engine for Open Source Code Supporting Structure-based Search," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, 2006.

[5] F. Behrang and A. Orso, "Test Migration Between Mobile Apps with Similar Functionality," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 54–65.

[6] F. Behrang and A. Orso, "Test Migration for Efficient Large-scale Assessment of Mobile App Coding Assignments," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[7] ——, "Test Migration for Efficient Large-scale Assessment of Mobile App Coding Assignments," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[8] F. Behrang, S. P. Reiss, and A. Orso, "GUIFetch: Supporting App Design and Development Through GUI Search."

[9] A. Caetano, N. Goulart, M. Fonseca, and J. Jorge, "JavaSketchIt: Issues in Sketching the Look of User Interfaces," in *AAAI Spring Symposium on Sketch Understanding*, 2002.

[10] W.-K. Chan, H. Cheng, and D. Lo, "Searching Connected API Subgraph via Text Phrases," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[11] S.-C. Chou, J.-Y. Chen, and C.-G. Chung, "A Behavior-based Classification and Retrieval Technique for Object-oriented Specification Reuse," *Softw. Pract. Exper.*, 1996.

[12] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?" In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15, Lincoln, Nebraska: IEEE Press, 2015, 429–440, ISBN: 9781509000241.

[13] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "WATER: Web Application TEst Repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, 2011.

[14] *Communicate with the UI thread*, `https://developer.android.com/training/multiple-threads/communicate-ui`, 2019.

[15] J. Creswell, *Educational Research: Planning, Conducting and Evaluating Quantitative and Qualitative Research*. Pearson, 2013.

[16] B. Daniel, Q. Luo, M. Mirzaaghaei, D. Dig, D. Marinov, and M. Pezzè, "Automated GUI Refactoring and Test Script Repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, 2011.

[17] D. Dig and R. Johnson, "Automated Upgrading of Component-based Applications," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, 2006.

[18] C. G. Drummond, D. Ionescu, and R. C. Holte, "A Learning Agent That Assists the Browsing of Software Libraries," *IEEE Transactions on Software Engineering*, 2000.

[19] M. Ermuth and M. Pradel, "Monkey See, Monkey Do: Effective Generation of GUI Tests with Inferred Macro Events," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016.

[20] *Espresso*, `https://developer.android.com/training/testing/espresso/`, 2019.

[21] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, 1996.

[22] M. Fazzini, E. N.D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests,"

in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, 2017.

[23] H. K. Flora, S. V. Chande, and X. Wang, *Adopting an Agile Approach for the Development of Mobile Applications*, 2010.

[24] C. Fu, M. Grechanik, and Q. Xie, "Inferring Types of References to GUI Objects in Test Scripts," in *2009 International Conference on Software Testing Verification and Validation*, 2009.

[25] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon, "SITAR: GUI Test Script Repair," *IEEE Transactions on Software Engineering*, 2016.

[26] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and Evolving GUI-directed Test Scripts," in *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[27] R. J. Hall, "Generalized behavior-based retrieval [from a software reuse library]," in *Proceedings of 1993 15th International Conference on Software Engineering*, 1993.

[28] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, 2014.

[29] D. Hemer and P. Lindsay, "Supporting Component-based Reuse in CARE," *Aust. Comput. Sci. Commun.*, 2002.

[30] G. Hu, L. Zhu, and J. Yang, "AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[31] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing GUI Test Suites Using a Genetic Algorithm," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, 2010.

[32] W. Janjic and C. Atkinson, "Leveraging software search and reuse with automated software adaptation," in *2012 4th International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE)*, 2012.

[33] W. Janjic, D. Stoll, P. Bostan, and C. Atkinson, "Lowering the barrier to reuse through test-driven search," in *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, 2009.

[34] J.-J. Jeng and B. H. C. Cheng, "Specification Matching for Software Reuse: A Foundation," in *Proceedings of the 1995 Symposium on Software Reusability*, 1995.

[35] M. G. Kendall, "A New Measure of Rank Correlation," *Biometrika*, 1938.

[36] S. Kumaresh and B. Ramachandran, "Mining Software Repositories for Defect Categorization," *Journal of Communications Software and Systems*, vol. 11, pp. 31–36, Apr. 2015.

[37] J. A. Landay and B. A. Myers, "Sketching interfaces: toward more human interface design," *Computer*, 2001.

[38] J. A. Landay and B. A. Myers, "Interactive Sketching for the Early Stages of User Interface Design," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1995.

[39] Q. Le and T. Mikolov, "Distributed Representations of Sentences and Documents," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, 2014.

[40] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes, "CodeGenie: Using Test-cases to Search and Reuse Source Code," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007.

[41] P. Li and E. Wohlstadter, "View-based Maintenance of Graphical User Interfaces," in *Proceedings of the 7th International Conference on Aspect-oriented Software Development*, 2008.

[42] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li, "ATOM: Automatic Maintenance of GUI Test Scripts for Evolving Mobile Applications," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[43] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining Android App Usages for Generating Actionable GUI-Based Execution Scenarios," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15, Florence, Italy: IEEE Press, 2015, 111–122, ISBN: 9780769555942.

[44] A. D. Lucia, R. Francese, G. Scanniello, G. Tortora, and N. Vitiello, "A Strategy and an Eclipse Based Environment for the Migration of Legacy Systems to Multi-tier Web-based Architectures," in *2006 22nd IEEE International Conference on Software Maintenance*, 2006.

[45] D. Lucredio, A. F. Prado, and E. S. de Almeida, "A survey on software components search and retrieval," in *Proceedings. 30th Euromicro Conference, 2004.*, 2004.

[46] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Transactions on Software Engineering*, 1991.

[47] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP Natural Language Processing Toolkit," in *Association for Computational Linguistics (ACL) System Demonstrations*, 2014, pp. 55–60.

[48] K. Mao, M. Harman, and Y. Jia, "Crowd intelligence enhances automated mobile testing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[49] L. Mariani, M. Pezzè, and D. Zuddas, "Augusto: Exploiting popular functionalities for the generation of semantic GUI tests with oracles," in *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[50] *Meet Android Studio*, `https://developer.android.com/studio/intro`, 2020.

[51] A. M. Memon, "Automatically Repairing Event Sequence-based GUI Test Suites for Regression Testing," *ACM Trans. Softw. Eng. Methodol.*, 2008.

[52] A. M. Memon and M. L. Soffa, "Regression Testing of GUIs," *SIGSOFT Softw. Eng. Notes*, 2003.

[53] N. Meng, M. Kim, and K. S. McKinley, "Systematic Editing: Generating Program Transformations from an Example," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, San Jose, California, USA: ACM, 2011, pp. 329–342, ISBN: 978-1-4503-0663-8.

[54] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," *CoRR*, 2013.

[55] R. Mili, A. Mili, and R. T. Mittermeir, "Storing and retrieving software components: a refinement based system," *IEEE Transactions on Software Engineering*, 1997.

[56] G. A. Miller, "WordNet: A Lexical Database for English," *Commun. ACM*, 1995.

[57] *Neo4j Graph Platform*, `https://neo4j.com`, 2020.

[58] T. A. Nguyen and C. Csallner, "Reverse Engineering Mobile Application User Interfaces with REMAUI (T)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015.

[59] A. Omari, S. Shoham, and E. Yahav, "Cross-supervised Synthesis of Web-crawlers," in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[60] ——, "Synthesis of Forgiving Data Extractors," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, 2017.

[61] *Pencil project*, `http://pencil.evolus.vn`, 2020.

[62] A. Podgurski and L. Pierce, "Retrieving Reusable Software by Sampling Behavior," *ACM Trans. Softw. Eng. Methodol.*, 1993.

[63] T. Pole and W. Frakes, "An Empirical Study of Representation Methods for Reusable Software Components," *IEEE Transactions on Software Engineering*, 1994.

[64] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding Plagiarisms among a Set of Programs with JPlag," *j-jucs*, 2002.

[65] X. Qi, F. Behrang, M. Fazzini, and A. Orso, "Identifying Features of Android Apps from Execution Traces," in *Under submission*, 2019.

[66] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall, "Mining Software Evolution to Predict Refactoring," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007, pp. 354–363.

[67] A. Rau, J. Hotzkow, and A. Zeller, "Efficient GUI Test Generation by Learning from Tests of Other Apps," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018.

[68] ——, "Transferring Tests Across Web Applications," in *Web Engineering*, T. Mikkonen, R. Klamma, and J. Hernández, Eds., 2018.

[69] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, 2010.

[70] S. P. Reiss, "Semantics-based Code Search," in *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[71] ——, "Seeking the User Interface," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014.

[72] E. S. Ristad and P. N. Yianilos, "Learning String-Edit Distance," *IEEE Trans. Pattern Anal. Mach. Intell.*, 1998.

[73] E. J. Rollins and J. M. Wing, "Specifications as search keys for software libraries," in *Proceedings 8th International Conference on Logic Programming*, 1991.

[74] M. de Sá, L. Carriço, L. Duarte, and T. Reis, "A Mixed-fidelity Prototyping Tool for Mobile Devices," in *Proceedings of the Working Conference on Advanced Visual Interfaces*, 2008.

[75] *SDK Platform Tools release notes*, `https://developer.android.com/studio/releases/platform-tools`, 2020.

[76] J. Seifert, B. Pfleging, E. del Carmen Valderrama Bahamóndez, M. Hermes, E. Rukzio, and A. Schmidt, "Mobidev: A Tool for Creating Apps on Mobile Phones," in *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, 2011.

[77] C Spearman, "The proof and measurement of association between two things," *International Journal of Epidemiology*, 2010.

[78] S. Staiger, "Static Analysis of Programs with Graphical User Interface," in *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, 2007.

[79] J. Starke, C. Luce, and J. Sillito, "Working with search results," in *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, 2009.

[80] V. Sugumaran and V. C. Storey, "A Semantic-based Approach to Component Retrieval," *SIGMIS Database*, 2003.

[81] S. Thummalapenta and T. Xie, "PARSEWeb: a programmer assistant for reusing open source code on the web," in *Proceedings ASE'07*, 2007.

[82] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of API methods from feature requests," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013.

[83] T. A. Vanderlei, F. A. Durão, A. C. Martins, V. C. Garcia, E. S. Almeida, and S. R. de L. Meira, "A Cooperative Classification Mechanism for Search and Retrieval Software Components," in *Proceedings of the 2007 ACM Symposium on Applied Computing*, 2007.

[84] *XML Path Language*, `https://www.w3.org/TR/xpath-30/`, 2019.

[85] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static Window Transition Graphs for Android," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[86] Y. Ye and G. Fischer, "Supporting Reuse by Delivering Task-relevant and Personalized Information," in *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[87] C. Zeidler, C. Lutteroth, W. Stuerzlinger, and G. Weber, "Evaluating Direct Manipulation Operations for Constraint-Based Layout," in *Human-Computer Interaction – INTERACT 2013: 14th IFIP TC 13 International Conference, Cape Town, South Africa, September 2-6, 2013, Proceedings, Part II*. Springer Berlin Heidelberg, 2013, pp. 513–529.

[88] S. Zhang, H. Lu, and M. D. Ernst, "Automatically Repairing Broken Workflows for Evolving GUI Applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013.